Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Monitorización de servidores Cloud: adquisición, procesamiento y envío de métricas.

Autor: Francisco José Ríos Bello

Tutor: Pablo Nebrera Herrera

Dep. de Ingeniería TelemáticaEscuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Monitorización de servidores Cloud: adquisición, procesamiento y envío de métricas.

Autor:
Francisco José Ríos Bello

Tutor:
Pablo Nebrera Herrera
Profesor asociado

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Proyecto Fin de Carrera: Monitorización de servidores Cloud: adquisición, procesamiento y envío de métricas.

Autor: Francisco José Ríos Bello

Tutor: Pablo Nebrera Herrera

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Presentando este proyecto pongo punto y final a una de las etapas más importantes en mi vida tanto a nivel personal como profesional. Este es el pequeño homenaje que tengo el honor de dedicarle a mis allegados, aquellos que han vivido y han sido partícipes de todo el esfuerzo que se ha dedicado en esta carrera.

En primer lugar es obligatorio y justo agradecer a mis padres, Paco y Salud. Ellos han sido los principales culpables de que se pueda concluir esta etapa de la manera que se hace. Siempre han estado dispuestos a proveerme de cualquier recurso que fuera necesario para poder llegar hasta aquí. En gran parte este logro es suyo.

También es importante recordar a mi abuela Josefa que seguro que estará orgullosa allá donde esté. Gracias también a mi abuelo Francisco, aunque no tenga formación técnica, es una fiel representación de la palabra "Ingeniero".

Agradecer también a mis tíos Isabel y Antonio, a mis tíos Paco y Rafi, y por supuesto también a mis primos María, Fran, Laura y Carlos.

No me olvido tampoco de mis amigos de Mairena, de Rául de Huelva y de mis amigos de la carrera, han sido un gran apoyo durante estos años. Por supuesto tampoco de mis compañeros de Netbeast con quien he compartido y seguiré compartiendo las ganas de aprender y trabajar.

Del mismo modo es necesario agradecer el apoyo a mi tutor Pablo Nebrera y a todo el equipo de Redborder por su contribución con el ambiente de trabajo en el que se ha desarrollado el proyecto.

Por último y no menos importante, agradecer a Asun. Ella siempre ha estado ahí en cada examen y trabajo para ayudarme a dar lo mejor de mi. Además ha sufrido en sus propias carnes mi imposibilidad de hacer muchos planes por tener que estudiar. Llegó el momento de poder realizar todos esos planes atrasados.

Francisco José Ríos Bello

Sevilla, 2016

Resumen

El punto de inicio de este proyecto se constituye bajo el deseo por parte de Redborder de tener un mayor control y conocimiento sobre qué ocurre dentro de sus instancias y su máquinas a nivel de servicios y recursos. El objetivo de esto no es otro que la validación de la correcta configuración y funcionamiento de los servicios que componen los manager de Redborder.

Para poder cumplir este objetivo se decide recurrir al desarrollo de un producto personalizado que presente obligatoriamente características dinámicas para que este case a la perfección con los principios de los entornos Cloud. Dado el partnership establecido entre Redborder y Cisco se pudo aprovechar la tracción a nivel profesional para poder tomar los requisitos y establecer los objetivos de una manera avanzada.

Valorando estos objetivos y los requisitos se llevó a cabo una investigación inicial con el fin de seleccionar un servicio de monitorización entre todos las existentes en el mercado. Esta concluyó con la elección de New Relic como plataforma con la que desarrollar el proyecto. Adicionalmente Cisco había aconsejado el uso de la misma cuando se le puso en conocimiento del proyecto que se iba a desarrollar.

La elección de New Relic y, en concreto su producto Plugin, trajo consigo un proceso de toma de requisitos y objetivos finales. Una vez se completó este punto se procedió a entrar en un ciclo de desarrollo-pruebas-validación con el fin no solo de cumplir los objetivos sino de hacerlo de una manera óptima.

A lo largo de todo el desarrollo del proyecto siempre se han tenido presentes una serie de premisas basadas en el carácter dinámico de la solución final (adaptabilidad al paradigma Cloud) así como la obtención de datos críticos sobre el funcionamiento de servicios y recursos. Esto ha tenido como consecuencia la finalización de un proyecto de gran utilidad que actúa como punto de entendimiento entre el equipo de desarrollo de software y el equipo de sistemas de Redborder. Es por eso que en varios puntos del proyecto se menciona el producto desarrollado como una herramienta fiel a la filosofía DevOps.

Abstract

The beginning of this project is set by Redborder's desire of a deeper control and knowledge over their IT infrastructures at service and resource levels. The aim is to validate the correct configuration of services comprised by the manager instances of Redborder.

In order to reach that goal, the decision of the development of a customized product was made. One of the main requirements is to preserve dynamic performance across the whole plugin to ensure compliance with Cloud environment principles. The goals and requirements in the scope of said project were defined as the partnership among Redborder on Cisco advanced. Taking those goals and requirements into account, the project commenced with an initial investigation to determine the monitoring platform or tool to be used. In the end, both by independent exploration and due to recommendation by Cisco, New Relic was chosen as starting base for the project. At this point, final requirements and goals were defined to begin the development-testing-verification cycle to ensure the optimal solution.

Long story short: dynamic performance was ensured while optimizing obtaining of critical data as main goal. This results in a project that perfectly fits the DevOps philosophy acting as a merging point between the software development team and the SysAdmin team in Redborder.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

IT	Information Technologies (Tecnologías de la información)
AWS	Amazon Web Services
SLA	Service Level Agreement
EC2	Elastic Compute Cloud
S3	Simple Storage Service
APM	Application Performance Monitoring
API key	Clave que identifica la cuenta de usuario a la que se va a reportar las métricas

1 INTRODUCCIÓN

"If someone asks me what cloud computing is, I try not to get bogged down with definitions. I tell them that, simply put, cloud computing is a better way to run your business."

- Marc Benioff, CEO of Salesforce-

En la actualidad del panorama de las tecnologías de la información han hecho aparición nuevos conceptos como el Big Data, IoT (del inglés Internet of Things) o Cloud Computing (Computación en la nube). Esta aparición invita a pensar si realmente se trata de tecnologías innovadoras o, si por el contrario, seguimos hablando de temas ya conocidos pero con nombres distintos.

Estos temas atraen a muchas empresas ya asentadas que, con el fin de ser más competitivos en el sector, intentan abrirse un hueco apostando por las tendencias anteriormente mencionadas. Además, cada año crece el número de startups de base tecnológica fundadas que encuentran una oportunidad de negocio en este escenario.

Investigando en los puntos en común que tienen estas startups se pueden encontrar patrones muy similares que evidencian y ratifican el hecho de que estas nuevas tendencias del sector constituyen una vuelta de tuerca más a la tecnología.

Concretando con el Cloud Computing no es extraño que se trate como un tema revolucionario. Esto se debe, en gran parte, a que ha cambiado la manera de desarrollar tanto a nivel de negocio como a nivel tecnológico la manera de crear nuevos productos y servicios por parte de la empresas del sector IT. Las ventajas que el Cloud Computing les ofrece a las nuevas startups y proyectos se basan en los conceptos de "pago por uso" y "computación elástica".

El término "pago por uso" hace referencia a la flexibilidad con la que se pueden usar los servicios Cloud de compañías como AWS(Amazon Web Services), Google o Microsoft. Esto evita. en la mayoría de los casos, la inversión en CPDs (Centros de Procesamiento de Datos) que se venía haciendo desde antaño.

Respecto al concepto de "computación elástica" se fundamenta en la posibilidad que tienen los servidores Cloud de adaptarse a nivel de recursos a la carga recibida "en vivo". Esto supone un plus en términos de alta disponibilidad para muchos servicios que lo precisan e incluso elimina muchos quebraderos de cabeza que ocasionaban los SLAs.

El objetivo de este proyecto es mostrar cómo la evolución de los sistemas e infraestructuras que el Cloud Computing trae consigo, acarrean también una evolución en los servicios y metodologías "clásicas". Concretamente se tratará el tema de la monitorización de servidores Cloud ya que el objetivos principal del proyecto es la realización de un plugin para Redborder en la plataforma de NewRelic. Con el fin de contextualizar el proyecto tanto a nivel teórico como práctico, se seguirá un orden lógico para abordar el desarrollo del plugin de una manera adecuada.

Se tomará como punto de partida la explicación del concepto de DevOps debido a que constituye la tendencia

con mayor crecimiento en términos del papel que los profesionales del sector IT juegan dentro de muchas compañías. Se continuará con una comparativa desde un punto de vista crítico entre los servicios de monitorización más utilizados y argumentando la elección de NewRelic por parte de Redborder. Posteriormente se explicarán las características y los servicios monitorizados dentro de la plataforma Redborder. A continuación se abordará con detalle el procedimiento, tecnologías y características implementadas en el plugin. Por último se explicarán otros servicios transversales que se ofrecen en la plataforma New Relic para cubrir un área mayor en el ámbito de la monitorización.

2 CONTEXTO Y COMPARATIVAS

"Las especies que sobreviven no son las más fuertes, ni las más rápidas, ni las más inteligentes; sino aquellas que se adaptan mejor al cambio. "

- Charles Darwin-

Fruto de la constante evolución del sector IT aparecen nuevos roles entre el conjunto de profesionales del mismo. En cierto modo se podría establecer una analogía con la Teoría de la Evolución de las Especies de Darwin. Mediante la observación de la demanda de nuevos profesionales del sector se puede comprobar qué características están tomando importancia.

Una de las filosofías metodológicas que está creciendo más en los últimos meses y años se identifica bajo el nombre de DevOps. Este no es más que un acrónimo del inglés Development and Operations (Desarrollo y Operación). A través de esta filosofía se abarcan flujos de trabajos que parten desde el desarrollo del software hasta la puesta en producción. Suele abarcar conceptos como los de integración continua, despliegue, etc… Además suele caracterizarse por favorecer la automatización de procesos.

Bajo esta filosofía emergente se pueden mover profesionales de características multidisciplinares. Esto se basa en la cooperación de los desarrolladores, administradores de sistemas, y aquellos que se encuentran en la frontera entre los dos anteriores y requieren de profundos conocimientos en ambas áreas para permitir que el ciclo de trabajo sea óptimo.

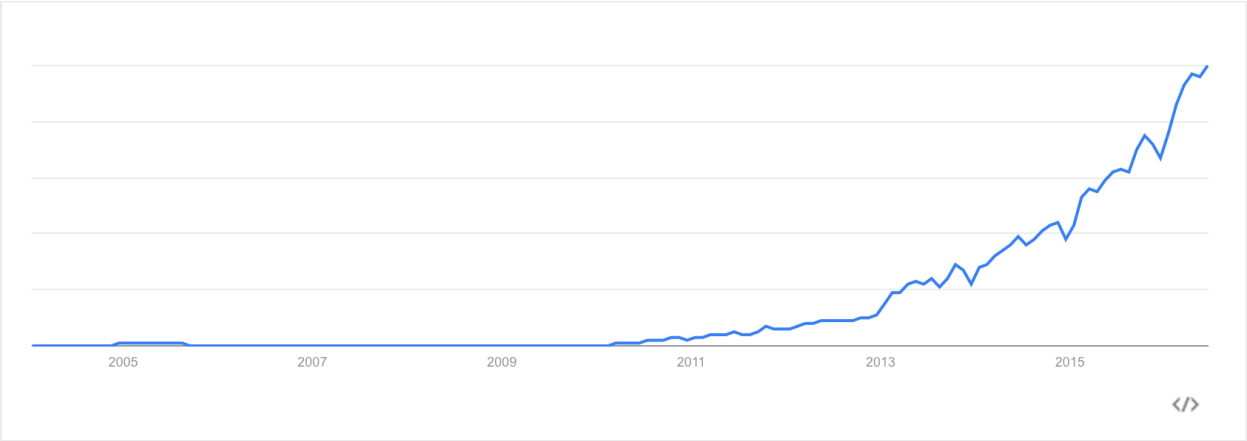

Figura 2-1 Tendencia del término DevOps (vía www.google.com/trends)

Son numerosas las herramientas de las que disponen los ingenieros centrados en este ámbito. Para poder medir la calidad de su trabajo es necesario que puedan acceder a métricas relevantes tanto a nivel de infraestructura como a nivel de servicios y de aplicación. Tradicionalmente se han usado herramientas para monitorizar redes, recursos en un servidor (CPU, memoria, etc…) pero eran poco frecuentes utilidades centradas a nivel de servicios y aplicación. Actualmente son más frecuentes los servicios de monitorización que abarcan servicios a

la vez que recursos propios de los servidores. Esta ampliación de rango sobre el que se reciben métricas permite identificar errores de desarrollo, despliegue y dimensionamiento de una manera más precisa.

New Relic se vende como una herramienta imprescindible en marcos de trabajo donde se sigue la filosofía DevOps. En este capítulo analizaremos el por qué de este enfoque y se comparará con las principales soluciones alternativas en temas de monitorización de servidores Cloud.

Antes de continuar con la comparativa aclararemos qué vamos a interpretar como servicios de monitorización. Nos referiremos al mercado de servicios de monitorización como aquellas principales soluciones que proveen a los profesionales IT de herramientas para conocer el estado de sus infraestructuras, servicios y recursos. Esta información les ayuda a sacar conclusiones acerca de la calidad de las soluciones, tanto a nivel de código como nivel operativo, que han desarrollado. Se valorarán aspectos como la presentación de la información, la cantidad de métricas que se reciben, la personalización de la información que se desea obtener, etc…

2.1. Principales servicios de monitorización

Previamente a analizar las características de New Relic se hará un repaso general sobre las soluciones que destacan en el sector de la monitorización. Las herramientas que se listarán a continuación han sido seleccionadas en base a su grado de utilización en el panorama actual. Además se ha mantenido un enfoque objetivo destacando las áreas que cubre cada uno de los servicios (infraestructuras/servidores, servicios/aplicaciones, etc…). También se ha valorado el hecho de la personalización de las métricas que manejan, los dashboards así como el acceso gratuito o de pago a sus distintas versiones.

2.1.1 AWS CloudWatch

Se trata de un servicio propio de AWS [1] mediante el cual se pueden crear dashboards que mostrarán las métricas que seleccionemos. Estas métricas son, en gran parte, estáticas y únicamente provienen de los servicios de AWS que se utilice (EC2, S3, …). Cada servicio propio de AWS tiene sus métricas y estas se pueden añadir a las gráficas con total libertad para el usuario.

Por defecto CloudWatch solo es capaz de monitorizar recursos previamente definidos. Si queremos enviar métricas personalizadas es necesario realizar solicitudes a una API.

Una funcionalidad que lo diferencia de otras soluciones es que ofrece la posibilidad de registrar eventos que actuarían como condicionante de una acción ejecutada por otros servicios de AWS. Por ejemplo: Se pueden ejecutar funciones de AWS Lambda que actualicen las entradas DNS cuando cambie el estado de una instancia (servidor de EC2).

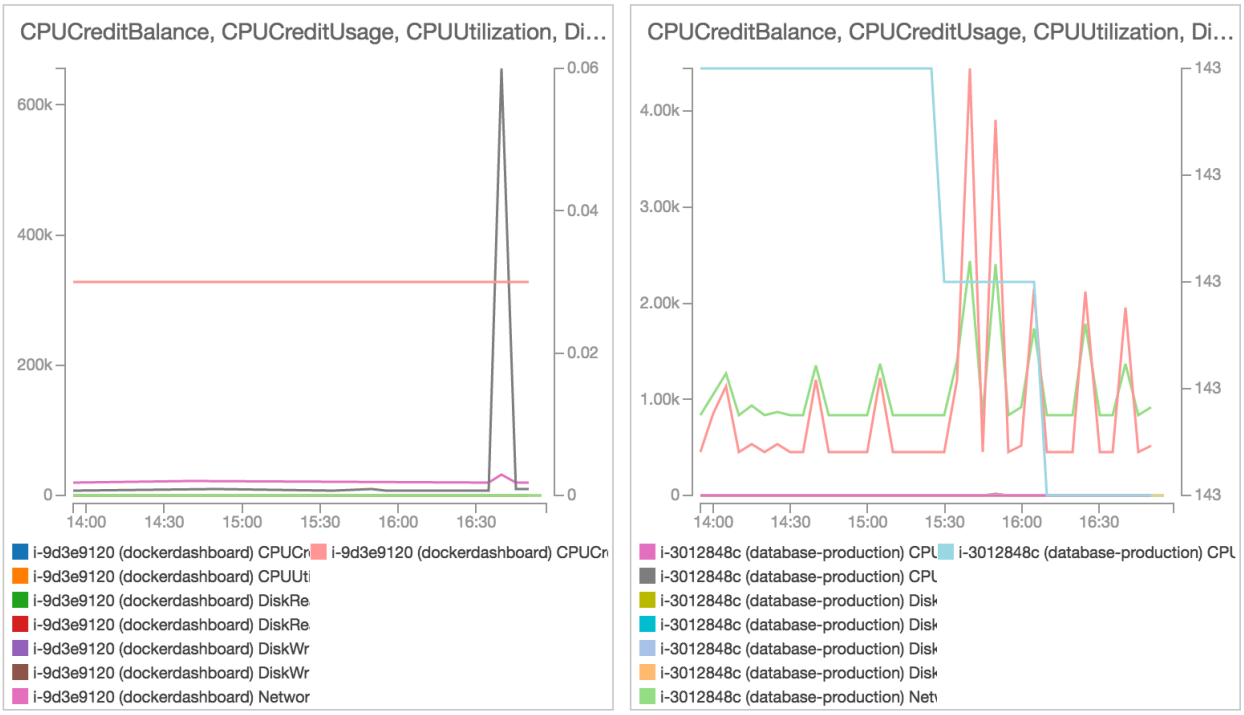

Figura 2-2 Dashboard de Amazon CloudWatch

2.1.2 Sensu

[2] En este caso el servicio está basado en Open Source aunque su modelo de negocio se apoye en un marco freemium. La instalación, configuración y dependencias debe ser gestionada por parte del usuario.

Se basa en el concepto de realizar chequeos del estado de las métricas configuradas por el usuario. Para poder realizar estos "checks" es necesario instalar las gemas correspondientes a los plugin que proporcionarán las funcionalidades deseadas. Sensu ofrece una gran variedad de plugins en su repositorio de github (https://github.com/sensu-plugins/). Además ofrece plantillas desarrolladas en ruby para la personalización de los chequeos.

La arquitectura se compone de clientes que reciben los checks desde un servidor. Esta comunicación se realiza a través de RabbitMQ. Cuando este obtiene la respuesta la almacena en un servicio de almacenamiento de estructuras de datos como es Redis.

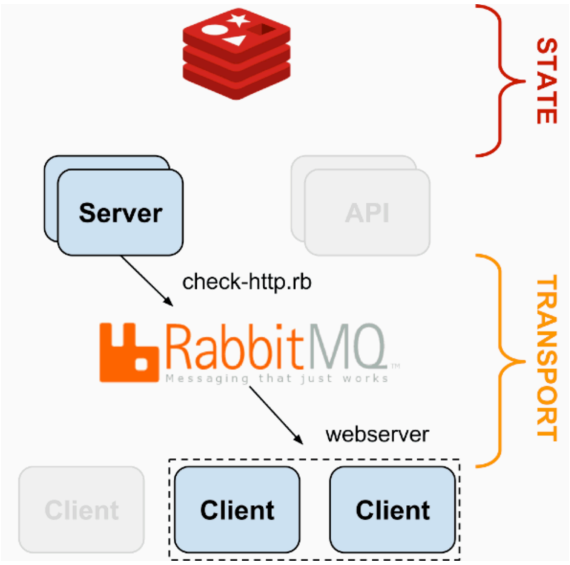

Figura 2-3 Arquitectura de Sensu

2.1.3 Datadog

A diferencia de las dos soluciones vistas anteriormente Datadog [3] ofrece la posibilidad de integrar otros servicios directamente y obtener las métricas a partir de ellos. Por ejemplo, para monitorizar una infraestructura desplegada en AWS únicamente es necesario introducir las credenciales del usuario de Amazon en el plugin de Datadog. Esto le aporta sencillez de instalación pero acompleja el proceso de envió de métricas personalizadas frente al procedimiento que se debe seguir en otras herramientas de monitorización.

Es importante destacar el alto grado de personalización de sus dashboards así como la amplia gama de servicios que permite integrar.

Como principal punto en contra se destaca el precio de la herramienta. A modo de prueba únicamente ceden una versión "trial" de 14 días de duración.

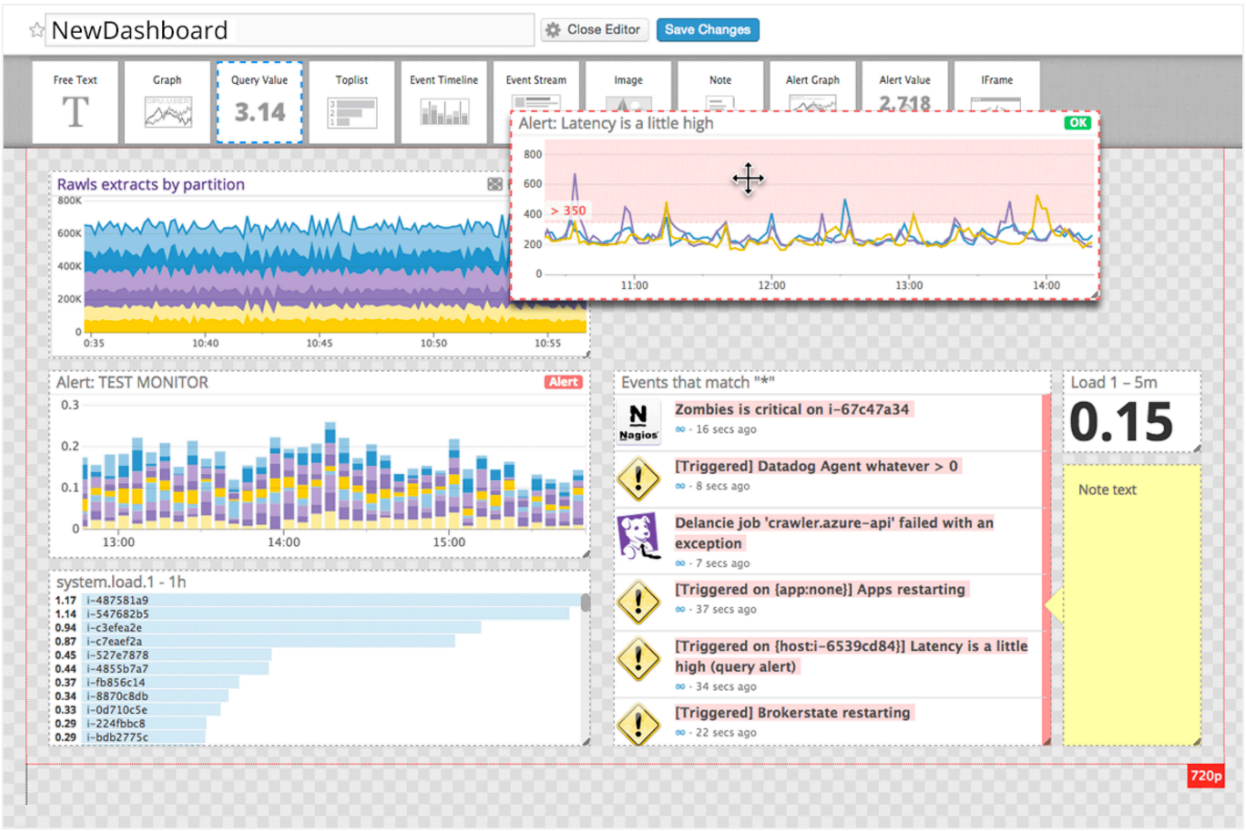

Figura 2-4 Dashboard de DataDog

2.1.4 Nagios

[4] Este servicio es uno de los que posee una trayectoria más larga en el ámbito de la monitorización. En este caso se centra principalmente en las métricas relativas a las infraestructuras IT. Posee tanto versiones OpenSource gratuitas como Enterprise de pago.

Respecto al tema de métricas personalizadas carece de un soporte tan completo como otras soluciones de sector. Cuenta con una amplia gama de productos con distintos objetivos entre los que destacan la monitorización de servidores, de red, de incidencias y de logs.

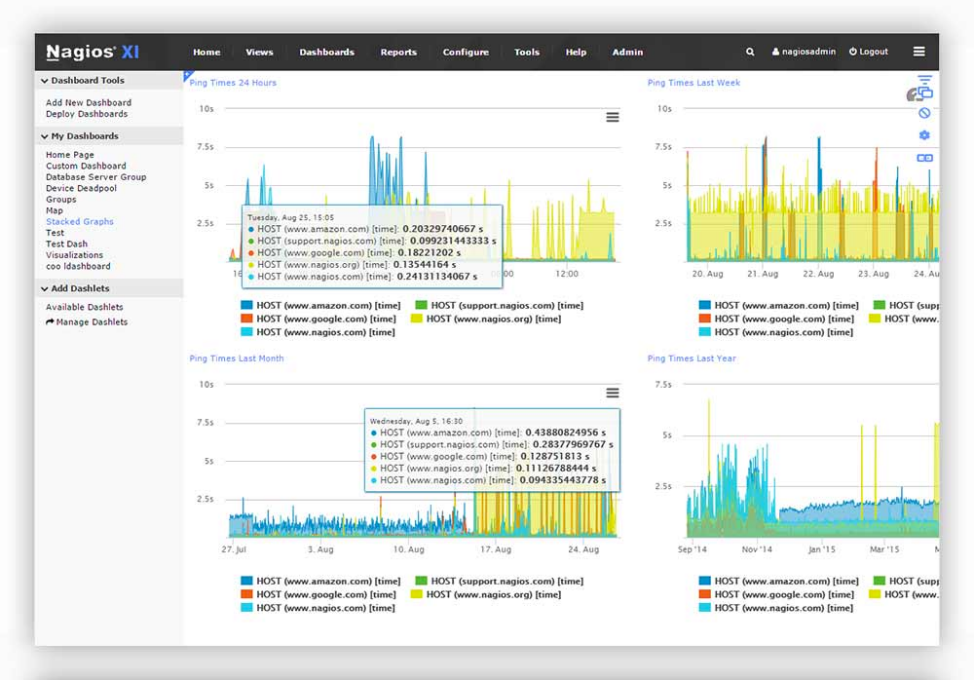

Figura 2-5 Dashboard de Nagios

2.1.5 Otros servicios

El conjunto de las anteriores herramientas analizadas pretende ser una muestra de las soluciones más conocidas en diversos ámbitos de la monitorización. No obstante a continuación se listarán sin entrar en mayor detalle otra soluciones clasificándolas por su objetivo principal:

Servidores e Infraestructura IT: Anturis, Boundary, CopperEgg, Munin, Solarwinds, Monit.

Monitorización de aplicaciones: AppDynamics, AppNeta, Boundary, Dynatrace, Instrumental, Solarwinds

Red: Munin, PRTG, Solarwinds.

2.2. Comparativa con New Relic

Como se ha señalado con anterioridad, Redborder apoyado por Cisco, decidió integrar una de las soluciones de New Relic para monitorizar con un rango más amplio la propia plataforma. En este apartado se expondrán las razones por las que se eligió esta herramienta y, concretamente, la solución de New Relic Plugin.

En las presentaciones de los productos que lideran el sector de la monitorización se valoraron ciertos aspectos que se tenían como requisitos antes de la toma de decisión de la herramienta a utilizar. Entre esas características destacan: La sencillez (tanto a nivel técnico como económico) con la que un usuario final accede a la herramienta, la personalización del dashboard, la cantidad de objetivos posibles a monitorizar y la libertad en el reporte de métricas personalizadas.

New Relic posee un catálogo amplio de productos para abarcar todos los ámbitos del sector de la monitorización. En el siguiente capítulo de esta memoria se profundizará sobre el producto de NewRelic Plugin y, en concreto cómo se ha realizado el plugin para Redborder. Además, en el último capítulo se explicarán y ejemplificarán el resto de soluciones. No obstante, ahora se va a argumentar el por qué de la elección de New Relic Plugin en base a los requisitos iniciales.

En primer lugar se requería una solución sencilla de instalar y sin coste alguno para el usuario final. En este tema New Relic Plugin permite usar plugins con dos sencillos pasos: Instalación del agente que va a reportar las métricas y configuración del API key. Esto lo hace más sencillo de usar que otras soluciones.

Es importante mencionar que New Relic soporta el desarrollo de plugins personalizados para optimizar las métricas y el dashboard del que dispondrá el usuario. Con esto cubrimos el requisito de las métricas

personalizadas. Además, el hecho de poder utilizar las métricas elegidas por el desarrollador, nos amplía el rango de objetivos a monitorizar. De nuevo tratamos un aspecto que hace de New Relic una solución preferible a otras.

Por último, otro requisito inicial era el dashboard personalizable. En este tema New Relic no es la solución que ofrece más libertad pero el grado en el que se cumplen el resto de requisitos frente a las otras herramientas disponibles hacen de la elección de NewRelic Plugin la opción óptima en términos generales.

3 COMPONENTES DE REDBORDER

No entiendes realmente algo hasta que seas capaz de explicárselo a tu abuela.

- Albert Einstein -

Para poder hacer una cobertura coherente y precisa de la plataforma Redborder se hace necesario un conocimiento previo de cuáles son sus componentes y su estructura. En primer lugar se analizará la plataforma desde un punto de vista general y seguidamente se analizarán por separado los servicios principales que han sido monitorizados.

3.1 Introducción a Redborder

La plataforma Redborder se centra en el ámbito de la seguridad y la monitorización de redes en tiempo real. Provee un gran control de la infraestructura y servicios presentes en la redes en las que se integra.

Su estructura general se basa en módulos divididos en base a la funcionalidad que aplican. De esta manera se consigue segmentar la solución ofrecida al cliente para optimizar la elección de los productos en base a sus necesidades:

3.1.1 Redborder IPS

Consiste en un sistema de prevención de intrusos (del inglés Intrusion Prevention System) que actúa conceptualmente como un control de acceso dentro de la red del cliente. Esta solución se ofrece en una topología basada en equipos que actúan bajo el rol de sensores que reportan eventos a un equipo con el rol de manager.

3.1.2 Redborder Flow

Bajo el propósito de monitorizar redes, Redborder flow se apoya en tecnologías como Netflow[1] y SNMP[2]. Ofrece al usuario información acerca del tráfico completo en la red en la que actúa.

Posee una alta escalabilidad en cuanto al tamaño de la red monitorizada debido a que aplica tecnologías basadas en BigData.

3.1.3 Redborder Malware

Este módulo se centra principalmente en la detección y en el bloqueo del malware. Gracias a su motor de detección permite analizar e identificar ficheros infectados así como evitar que se extienda esta infección.

3.1.4 Redborder Social

Se trata de un componente enfocado a temas más comerciales que a la seguridad. Mediante el uso de Redborder Social el cliente puede percibir cuál es la opinión que los usuarios de un producto o servicio determinado están compartiendo en las redes sociales.

[1] Netflow es un protocolo desarrollado por Cisco centrado en la monitorización de red.
[2] SNMP (del inglés Simple Network Management Protocol) es un protocolo de gestión de red.

Como ejemplo se puede mencionar su uso durante el Mobile World Congress[3] de Barcelona para detectar qué es lo que los usuarios de la red WiFi del evento comentaban acerca de esta en Twitter.

3.2 Estructura general

Como se ha comentado con anterioridad, la plataforma se apoya en dos tipos de equipos. Estos son el **manager** y el **sensor**. Ambos basados en distribuciones Linux CentOS[4].

3.2.1 Sensor

Son los equipos instalados dentro de la red del cliente. Se encargan de reportar datos y eventos al manager. Además pueden actuar directamente sobre la red si son configurados para tal efecto. Por ejemplo podrían cortar una conexión en la que se ha detectado un evento malicioso actuando como IPS.

3.2.2 Manager

Se concibe como el cerebro principal de un escenario en el que opera la plataforma. Procesa y analiza la información reportada desde los sensores e incluso la muestra en el dashboard de la interfaz web.

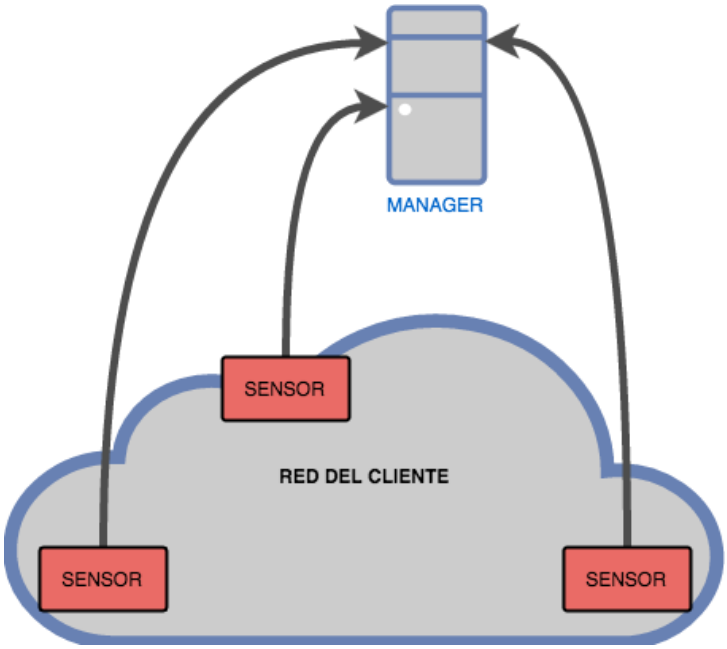

Figura 3-1 Escenario genérico usando Redborder

3.3 Redborder Manager

En capítulos anteriores se ha mencionado como objetivo del proyecto la monitorización de la plataforma Redborder. Concretando más, se toma como punto inicial la imagen perteneciente al manager del Redborder. El principal motivo que lleva a esta decisión es el rol de nodo crítico que juega dentro de un escenario de

[3] El Mobile World Congress (MWC) es un evento anual celebrado en Barcelona en el mes de febrero. Congrega a una gran cantidad de asistentes atraídos por las empresas más punteras del sector tecnológico.
[4] CentOS es un sistema operativo derivado de Red Hat Enterprise Linux.

sensores y un manager. Además es de destacar su complejidad a nivel de servicios que posee. En los siguientes apartados de este capítulo se analizarán los recursos y servicios que han sido objeto de monitorización. No obstante, a continuación se muestra un esquema general de los componentes de un manager con el fin de contextualizar los servicios monitorizados.

3.3.1 Componentes

Concretando más sobre el ámbito que ataca el plugin desarrollado se entra en los componentes a nivel de servicios del manager de Redborder.

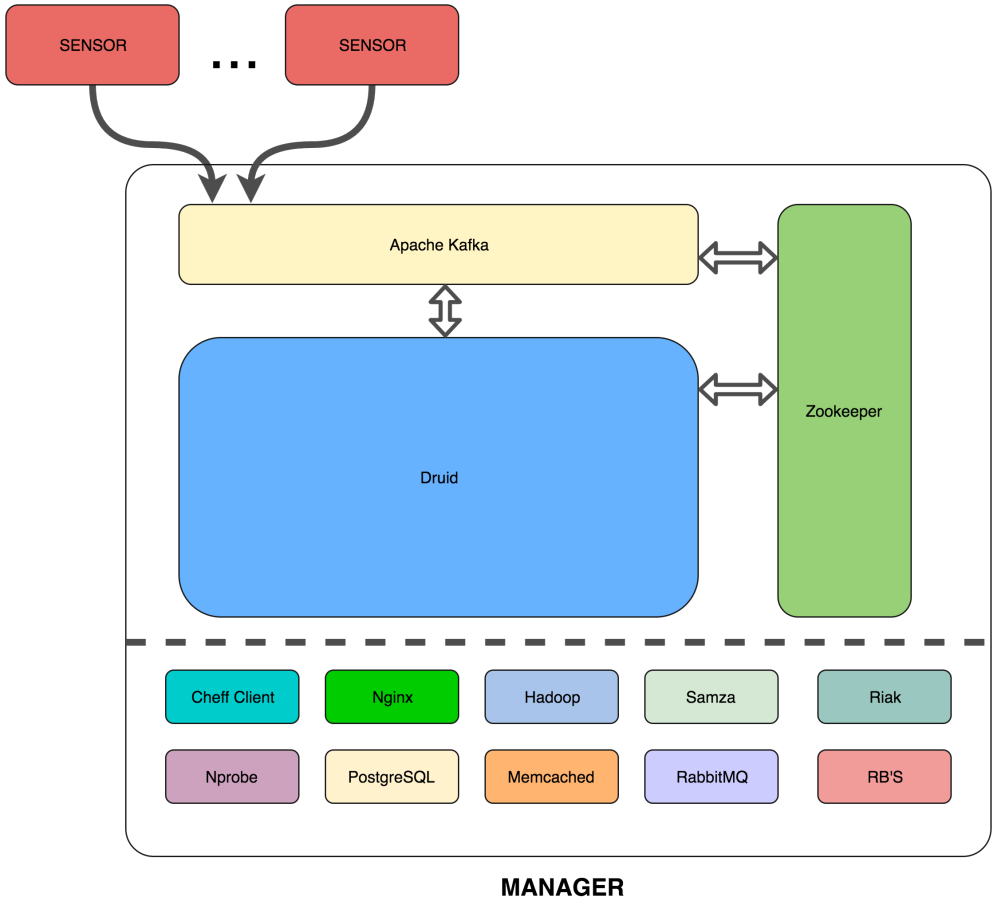

Figura 3-2 Servicios del manager de Redborder

3.3.2 Redborder Live

Recientemente Redborder ha lanzado un producto bajo el nombre de Redborder Live. Principalmente comprende una serie de funcionalidades y productos ya servidos por Redborder. La principal característica de este nuevo producto es su despliegue en la Cloud.

A lo largo de la explicación de los componentes de la platforma Redborder no se ha mencionado hasta ahora la Cloud a pesar de que el proyecto esté enfocado a este ámbito. Por ello es importante destacar que aunque Redborder ofrezca este nuevo producto basado en la Cloud, los servicios objetivo a monitorizar siguen siendo los mismos.

Cabe destacar también que el producto Redborder Live posee una dinamicidad destacable frente a los ofrecidos anteriormente. Es en este punto donde se fundamenta que el plugin desarrollado esté enfocado a servicios Cloud a pesar de que sea completamente funcional en servicios de tipo On-premise.

3.4 Apache Kafka

Apache define Kafka [5] como un servicio particionado y orientado a la replicación cuyo propósito principal es proveer un sistema de cola de mensajes. Es muy utilizado en situaciones en las que es necesario comunicar flujos de información entre varias aplicaciones.

Utiliza una arquitectura de sistema distribuido basándose en los conceptos de publicador y subscriptor. Los mensajes que comprenden la información compartida se categorizan en topics. Los procesos publicadores de estos mensajes se conocen con el nombre productores de topics. Análogamente, los subscriptores son los que consumen estos topics. Además, Kafka se ejecuta en clústeres compuestos de uno o varios servidores de Kafka donde cada uno de ellos se conoce como broker.

Para funcionar, Kafka hace uso de un protocolo propio basado en TCP y se apoya en Apache Zookeeper para almacenar el estado de los broker. En la siguiente figura se muestran los principales servicios con los que interactúa Kafka dentro de una configuración básica de Redborder.

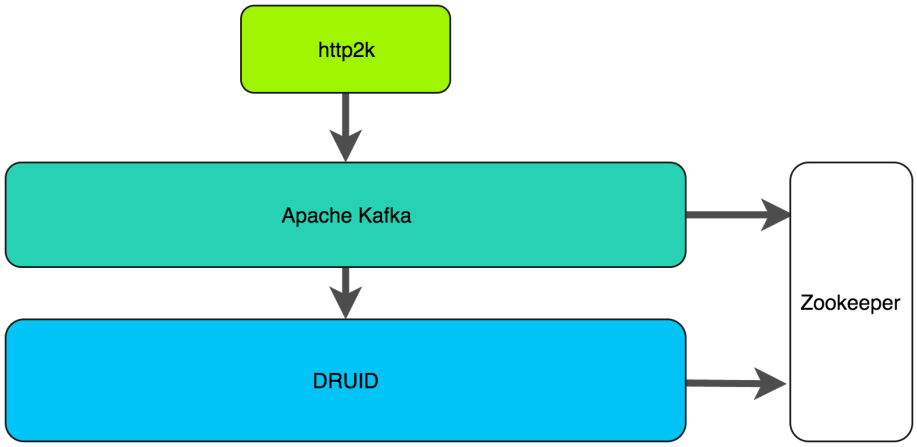

Figura 3-3 Kafka en Redborder

Como se puede comprobar, http2k es un servicio publicador de información recibida desde los sensores de Redborder. Druid en este caso actúa de consumidor de Kafka, en el apartado 3.6 se abordará con mayor detalle el funcionamiento de este servicio.

3.5 Apache Zookeper

Apache Zookeeper [6] es un servicio centralizado para almacenar y gestionar información relativa a la configuración y a la sincronización de aplicaciones distribuidas. Básicamente actúa como punto de coordinación para servicios distribuidos que cooperan entre sí. Además confiere características como la alta disponibilidad y una alta tolerancia ante posibles fallos gracias a la redundancia.

Para poder coordinar esta redundancia a nivel de nodos servidores se tiene el rol de líder. Este es elegido automáticamente por Zookeper. El resto de nodos son una copia de este.

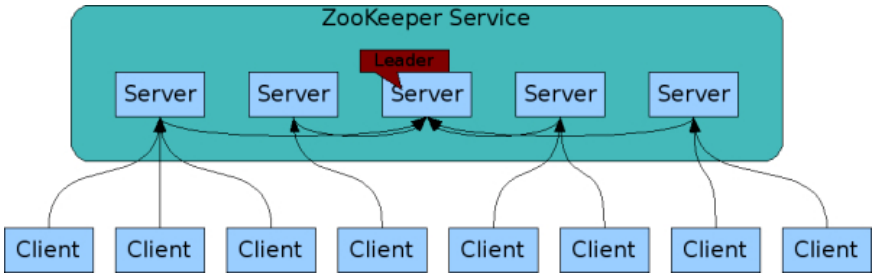

Figura 3-4 Estructura de Zookeeper

3.6 Druid

Druid [7] es un sistema de almacenamiento y análisis de datos basado en código libre. El objetivo principal de Druid es ofrecer una herramienta de consulta de un gran volumen de datos en tiempo real. Para ello implementa características como la ingestión de datos con baja latencia y la exploración flexible y agrupamiento rápido de datos.

3.6.1 Conceptos

Para manejar los datos Druid los agrupa en colecciones conocidas como segmentos. Un segmento contiene datos y eventos generados en un determinado intervalo de tiempo.

La arquitectura se basa en clústeres con distintos tipos de nodos. Cada tipo de nodo realiza un determinado conjunto de tareas. Los tipos de nodos son Historical, Broker, Coordinator y Real-time. En las siguientes subsecciones se comentará qué papel tiene cada uno dentro de un clúster de Druid.

Para el funcionamiento de Druid se precisa de una serie de dependencias. Zookeper funciona paralelamente para la comunicación entre nodos en un clúster. Además se requiere de un sistema de almacenamiento de metadatos para el que se puede usar MySQL o PostgreSQL[5]. La última dependencia de la que precisa Druid es la recogida bajo el nombre de Deep Storage. Esta tiene por objetivo actuar como un sistema de copias de seguridad permanente de segmentos y para ello se puede usar S3 o HDFS[6].

3.6.2 Nodos Historical

Druid define el rol de Historical como la espina dorsal de un clúster. Son los encargados de extraer información de los segmentos relativos a una consulta ocurrida. Para ello descarga los metadatos que localiza comunicándose con los nodos Coordinator a través de Zookeeper dado que no mantiene conexión directa con otros nodos. Esta descarga solo se produce si la información sobre los segmentos no ha sido almacenada en caché debido a una consulta previa.

3.6.3 Nodos Realtime

Los nodos de tipo Realtime son los encargados de generar los segmentos a partir de los datos que van obteniendo, por ejemplo, a través de Kafka. Estos segmentos los almacenan en el Deep Storage y la transferencia es controlada por Zookeeper.

Además los nodos Realtime usan la base de datos (MySQL o PostgreSQL) para almacenar metadatos.

3.6.4 Nodos Broker

Este tipo de nodos se encarga de manejar las consultas realizadas a un cluster. Para ello hace uso de los metadatos almacenados en la base de datos a los que accede a través de Zookeeper. Mediante la interpretación

[5] MySQL y PostgreSQL son sistemas de gestión de bases de datos relacionales.
[6] HDFS o Hadoop Distributed File System es un sistema de ficheros distribuidos.

de estos metadatos, es capaz de identificar y localizar los segmentos.

Para disminuir la latencia se hace uso de una caché. De este modo cuando se consultan datos que ya había sido requerido previamente, se puede servir directamente sin necesidad de volver a localizar los segmentos en el Deep Storage.

3.6.5 Nodos Coordinator

Su papel principal es la gestión y distribución de los segmentos. Bajo tal fin, le comunica a los nodos historical (a través de Zookeeper) qué segmentos cargar y cuales descartar. Además, los nodos Coordinator se encargan de gestionar la replicación de segmentos y el balanceo de carga. Por ello tienen que estar en comunicación constante con Zookeeper para conocer el estado del cluster.

3.7 Nginx

Nginx [8] es el servicio utilizado para ofrecer el dashboard a través de una interfaz web entre otras tareas. Se trata de un proyecto de código libre que provee de funcionalidades de servidor web, reverse proxy y balanceador de carga entre otras. Su uso está creciendo bastante frente al del conocido servidor web Apache dado que Nginx es bastante más ligero y sencillo de configurar.

Su comportamiento ante un número alto de clientes es otro de los principales argumentos que llevan a muchos servidores a sustituir a Apache.

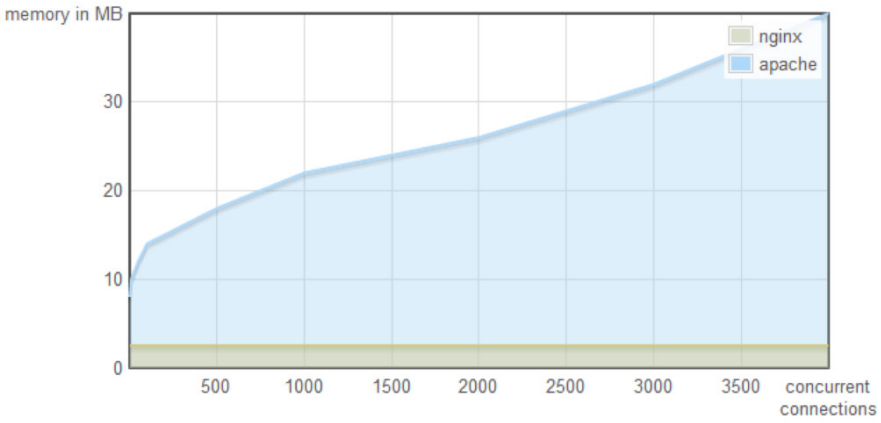

Figura 3-5 Comportamiento de Nginx y Apache frente a conexiones concurrentes[7].

3.8 PostgreSQL

En el apartado 3.6 se mencionó el servicio de gestión de bases de datos PostgreSQL [9]. Este es un proyecto de código libre soportado por una comunidad de desarrolladores denominada PGDG (PostgreSQL Global Development Group).

Entre las principales características podemos destacar la alta concurrencia que ofrece. Esto se debe al método que usa para gestionar las versiones de los datos que almacena. Este método se conoce bajo el nombre de MVCC (Control de Concurrencia Multiversión). A efectos prácticos si un proceso escribe en una tabla, otros pueden acceder a la misma sin que se produzca un bloqueo.

[7] Se conoce como conexión concurrente a la ejecución de una tarea por parte de un cliente ante un servidor en un momento determinado a la vez que otros usuarios lanzan otras tareas en el mismo servidor. Por ejemplo, si 10 usuarios acceden a una página alojada en un servidor web a la vez, se contaría como 10 conexiones concurrentes.

3.9 Chef

Chef es una que permite desplegar y gestionar la configuración de servicios en servidores y otros equipos de manera automática. Es uno de los más utilizados en el entorno empresarial del sector IT dadas ciertas características que lo hacen preferible frente a soluciones similares como Ansible o Puppet[8].

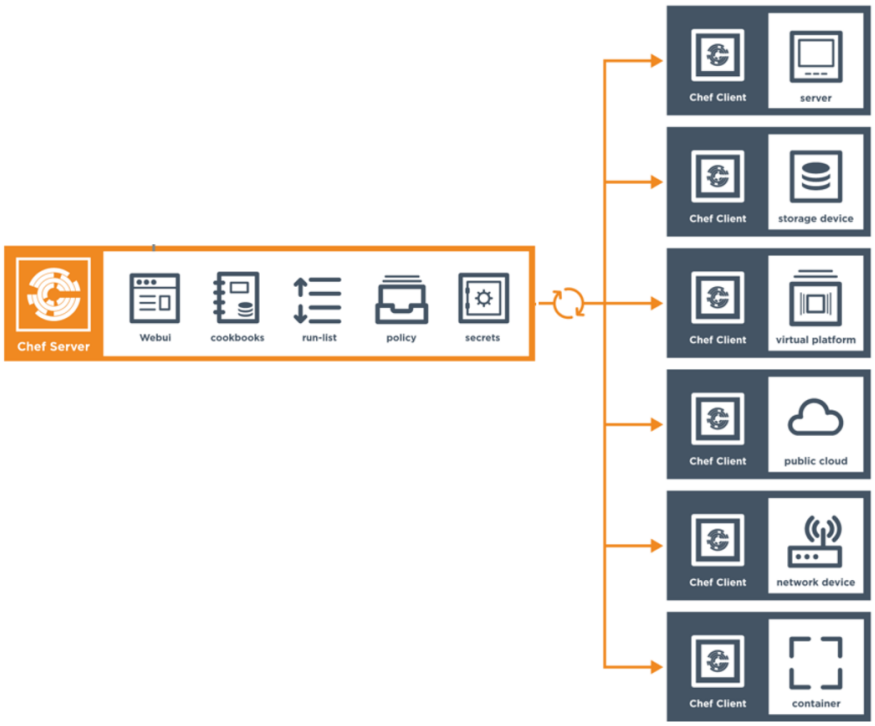

Figura 3-6 Estructura básica de Chef en funcionamiento

En la Figura 3-6 Estructura básica de Chef en funcionamiento se muestra la estructura básica que permite inducir la configuración desde el Chef-Server hasta los nodos a gestionar/configurar. El conjunto de configuraciones que se aplicarán a los distintos nodos se recogen en ficheros llamados receipes (que significa recetas en inglés), ficheros de configuración, definiciones, atributos y otros recursos. Todos ellos se agrupan en cookbooks (se traduce como recetario en inglés).

Un nodo bajo la gestión de Chef debe tener instalado un cliente Chef que será el encargado de sincronizar los cookbooks y realizar las tareas asociadas a dejar el nodo en el estado deseado. Los tipos de nodos gestionados son bastante diversos pues se pueden gestionar desde servidores (tanto físicos como alojados en cloud) hasta dispositivos de red (routers y switches de Juniper, Cisco, Arista, etc…) pasando por contenedores (Docker, Rocket, etc…).

Un servidor de Chef no es más que un servicio ejecutado en un nodo que se encargará de gestionar las configuraciones y tareas disponibles para aplicar en los nodos. Los administradores de sistemas y DevOps son los encargados de proveer al servidor los coobooks y demás información asociada a través de la interfaz web.

[8] Ansible (https://www.ansible.com/) y Puppet (https://www.puppet.com/) son herramientas de gestión de configuración automática para servidores.

3.10 Hadoop

Apache Hadoop [10] es un framework que permite el procesamiento distribuido de grandes cantidades de datos a través de clústeres de equipos usando modelos de programación simple. Su diseño está enfocado a escalar desde pocos servidores hasta cientos de ellos ofreciendo cada uno almacenamiento y procesamiento.

Hadoop incluye principalmente de cuatro módulos:

- **Hadoop Common:** Provee la utilidad principal para integrar el resto de módulos.
- **HDFS (Hadoop Distributed File System):** Se trata de un sistema de ficheros distribuidos que permite una alta disponibilidad y un acceso rápido a datos almacenados.
- **Hadoop YARN:** Es un framework para la programación de tareas y la gestión de recursos de los clústeres.
- **Hadoop MapReduce:** Sistema basado en YARN que tiene como objetivo el procesamiento paralelo de grandes cantidades de datos.

3.11 Otros servicios

El manager de Redborder incorpora muchos más servicios para realizar las funcionalidades objetivo. Se omite la explicación de estos dado que no constituyen puntos tan críticos como los explicados en apartados anteriores en una configuración genérica del manager.

4 DESARROLLO DEL PLUGIN

Los ordenadores son inútiles. Solo pueden darte respuestas.

- Pablo Picasso -

Con este capítulo se entra ya en materia práctica de la solución llevada a cabo para Redborder con el fin de reportar métricas que abarquen toda la plataforma a nivel de recursos y a nivel de servicios. Como bien se ha ido introduciendo a lo largo de este capítulo se ha hecho uso de la solución de Plugin personalizado de New Relic [11] para el objetivo propuesto. En las siguientes líneas se detallarán ciertos aspectos y características de esta solución para proseguir con la explicación del proceso de desarrollo.

En el capítulo 3 se expuso una serie de argumentos que hacían de New Relic Plugin la solución óptima en base a los requisitos presentados por Cisco y Redborder para monitorizar los nodos que integran la plataforma de este último. New Relic es más conocido por sus productos APM (Application Performance Monitoring) y Synthetics Monitoring sobre los que se explicará algunos detalles en el capítulo 6 de este documento. En ese mismo capítulo también se tratarán servicios como Server Monitoring. No obstante, ahora es el turno de New Relic Plugin y, concretando, el Plugin de Redborder dado que este es el principal componente del eje del proyecto.

La solución que New Relic ofrece al usuario a través de los Plugin se basa principalmente en un agente específico para cada cada servicio, stack [9] o plataforma. Este agente reporta las métricas a la plataforma de New Relic desde el nodo en el que se ejecuta. En la plataforma el usuario final visualiza estos datos mediante los dashboard prediseñados para el plugin que corresponde al agente.

[9] Se conoce como stack a una pila de tecnologías o servicios que funcionan de manera conjunta para ofrecer una funcionalidad más genérica a partir de las funcionalidades más específicas de los mismos. Por ejemplo LAMP stack (Linux, Apache, MySQL y PHP/Python/Perl) es muy usada para servidores web. Otro ejemplo más de moda es MEAN stack (MongoDB, ExpresJS, AngularJS y NodeJS).

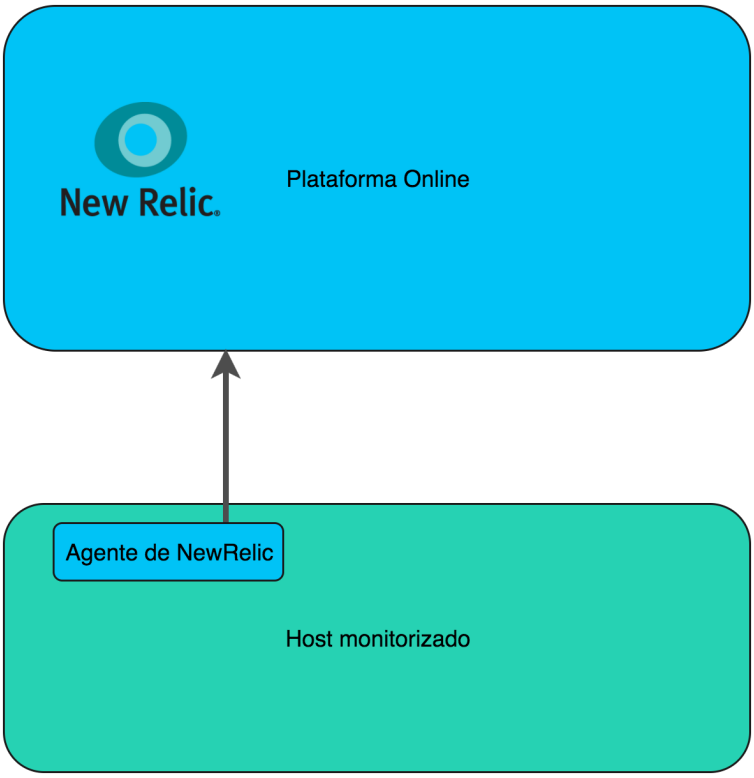

Figura 4-1 Esquema básico de New Relic Plugin

La instalación y configuración del plugin se produce en el nodo que va a reportar, en el agente. Cada desarrollador de un plugin tiene la libertad de diseñar estos pasos iniciales de manera personalizada. Generalmente, las instrucciones de instalación se basan en la descarga del código fuente del plugin y la inserción del API Key (también llamado License Key) dentro de un fichero de configuración incluido en el código descargado. Otro de los principales modos de instalación de un agente se basa en el llamado NPI (New Relic Platform Installer). Este consiste en en una aplicación de NodeJs que permite descargar, instalar y ejecutar algunos plugin de la plataforma.

Para la ejecución del agente existen también diferencias en función, principalmente, del lenguaje usado en su desarrollo. Ruby y Java son los más frecuentes. Una vez se pone en marcha el agente con el API Key configurado, en la plataforma New Relic se añade el plugin automáticamente a la cuenta asociada a dicha clave. Desde ese momento ya podemos visualizar las métricas en los distintos dashboards configurados en el plugin.

Todos los nodos que tengan un agente instalado y configurado con la misma clave reportarán al mismo plugin en la misma cuenta. Esto provocará la visualización de las métricas las mismas gráficas de los dashboards. Se trata de un aspecto importante a la hora de diseñar un dashboard para un plugin determinado pues hay que tener en cuenta que las gráficas que compongan el dashboard deben contener un número de datos que permita extraer la información de manera rápida.

En la última sección de este capítulo se tratará el diseño del dashboard para el plugin de Redborder. Antes se abordará con un alto nivel de detalle el proceso de desarrollo del agente.

4.1 Agente: Introducción y estructura

El agente correspondiente al plugin de Redborder ha abarcado la mayoría del tiempo empleado para este proyecto. Constituye la parte más importante del plugin ya que en él se obtienen, procesan y envían las métricas que serán indexadas en la plataforma New Relic.

En el capítulo 3 se ha abordado la estructura principal de Redborder a nivel de productos ofrecidos. Además se

introdujeron los principales servicios que integran el nodo principal de cualquier solución de Redborder, el manager. Todo ello es fruto de una investigación inicial que fue necesaria para conocer los objetivos a monitorizar. En este apartado se analizará la obtención y el posterior procesado de las métricas por cada servicio, conjunto de servicios o recurso. Entendiendo como recurso aspectos comunes (CPU, memoria, red, etc…) de una instancia en la Cloud, máquina virtual o equipo físico.

Con el fin de abordar estos aspectos de una manera más práctica se seguirá un esquema basado en subapartados que llevarán como título el ámbito o servicio del que toman las métricas. Dentro de cada subapartado se explicarán las métricas reportadas así como la obtención de las mismas desde un punto de vista práctico y haciendo referencia a código fuente y algoritmos. Además se incluirá también una breve sección que explicará el proceso de validación de la obtención de la o las métricas así como de su procesamiento.

4.1.1 Componentes estructurales y funcionales

Como punto de inicio en la explicación del agente desarrollado se van a listar una serie de características estructurales y funcionales importantes del agente del plugin.

En primer lugar es necesario comentar que se inició el desarrollo haciendo uso del SDK de Ruby. El principal punto a favor que motivó esta elección fue la conservación de la simetría a nivel de lenguajes de la plataforma Redborder dado que la gran mayoría de los componentes se han desarrollado en este lenguaje.

La estructura a nivel de ficheros que integran el agente se explicará a continuación haciendo uso de un diagrama:

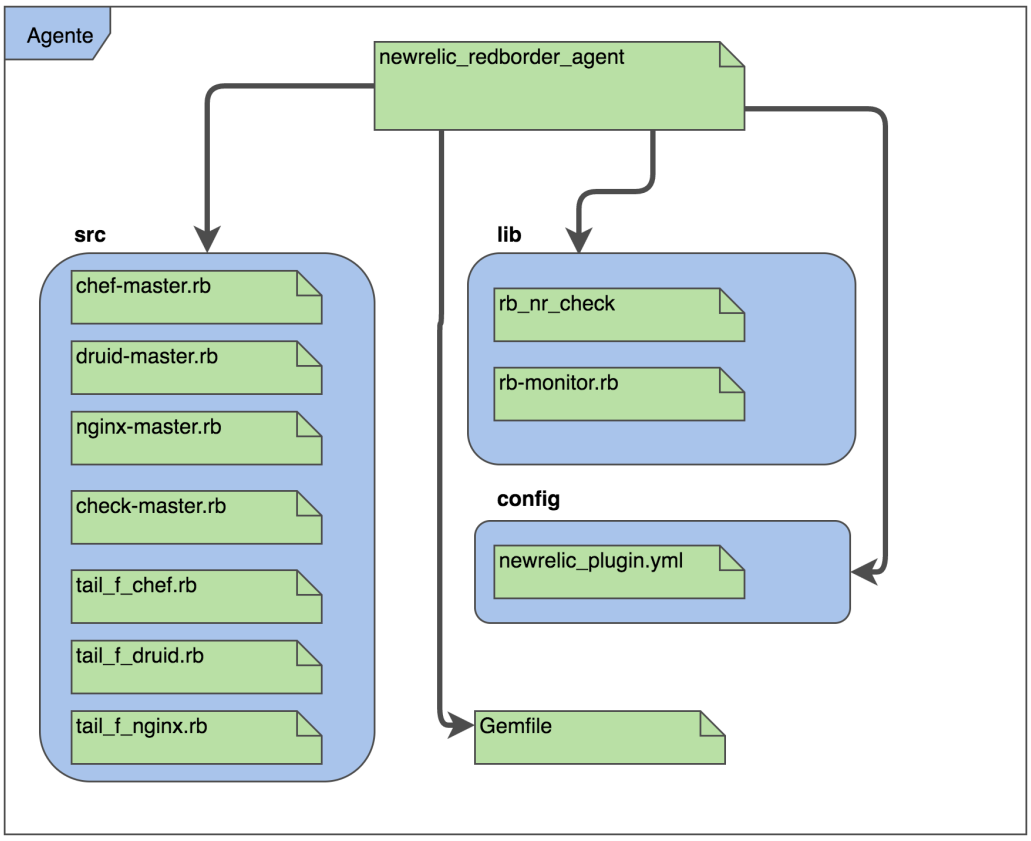

Figura 4-2 Estructura de ficheros

Como se puede comprobar en la Figura 4-2 Estructura de ficheros, el fichero principal es *newrelic_redborder_agent*. Este hace uso de otros ficheros de propósitos más específicos para reportar las métricas hacia la plataforma de New Relic. Estos ficheros se irán comentando en este capítulo a medida que se vaya avanzando en la explicación de los servicios y recursos monitorizados. En este subapartado únicamente se abordará el fichero principal y el de configuración.

El inicio del agente parte de una base genérica especificada por New Relic, esta es el SDK de Ruby. Para poder comenzar a utilizarlo es necesario incluir la gema que incorpora los recursos necesarios. Para ello se añadió una línea con el contenido `gem 'newrelic_plugin'` al fichero *Gemfile*. En él se incluyen todas las dependencias propias de un proyecto en Ruby.

Estas dependencias también se deben especificar en el fichero principal para que los recursos puedan ser cargados con el fin de utilizarlos. Para ello se añadió la línea `require 'newrelic_plugin'` al inicio del fichero *newrelic_redborder_agent*. El resto de dependencias que fueron surgiendo a lo largo del desarrollo (snmp, pry, file-tail, rationalist) se añadieron con el mismo procedimiento.

Habiendo completado el procedimiento relativo a las dependencias de NewRelic, ya se comenzó a desarrollar la base que permitiría reportar métricas hacia la plataforma. La estructura del agente se construyó empaquetándolo en un módulo tal y como recogía la documentación. Este módulo recibe el nombre de `RedborderAgent`. En él se incluye la clase `Agent` que hereda de `NewRelic::Plugin::Agent::Base`. Esto permite que se configure y ejecute el plugin en base al fichero de configuración y mediante dos funciones provistas con el SDK que se mencionarán a continuación.

Dentro de la clase Agent se configuraron ciertos parámetros necesarios para iniciar el agente:

```
agent_guid 'com.redborder.redborder'
agent_version '1.0.2'
agent_config_options :snmp_host, :snmp_community, :services
agent_human_labels('Manager') { 'testingCluster' }
```

El primero de ellos se corresponde con el identificador único gobal (GUID, del inglés Global Unique Identifier). El formato para darle valor es 'com.*nombre_compañía.nombre_plugin*'. Es importante que este parámetro sea único y que no sea modificado una vez publicado el plugin. La principal razón es porque los datos de los usuarios y la interfaz o el dashboard se asocian a este parámetro dentro de la plataforma New Relic Plugin. En el caso de que este sea modificado, en la plataforma dejará de aparecer el dashboard creado para el nombre anterior así como los datos reportados.

Otro de los parámetros configurados se corresponde con la versión del plugin. Esto no es más que un indicador para el usuario final para que este esté informado de si la versión del agente que está usando está actualizada o no. Este parámetro sigue el estándar de Semantic Versioning. Este se basa en tres campos separados por un punto. El formato es MAJOR.MINOR.PATCH:

- MAJOR: Se incrementa cuando se realizan cambios importantes en la API que provocan incompatibilidades con versiones anteriores.
- MINOR: Se incrementa al añadir funcionalidades adicionales.
- PATCH: Se incrementa cuando se hacen modificaciones relativas a solventar bugs o cambios poco significativos.

El siguiente parámetro listado permite utilizar dentro del agente variables configurables en el fichero de configuración newrelic_plugin.yml. En el caso del agente desarrollado se han configurado parámetros relativos a SNMP y un array con los servicios sobre los que se ejecutarán checks.

Por último se añadió un parámetro para poder tener tantos dashboards como clusters reportan. Dentro de la interfaz web de usuario en el plugin de Redborder veremos tantas instancias como clusters se han configurado para reportar. Esto permite al usuario final tener organizados los dashboards y no mezclar datos indeseados en las gráficas.

Una vez se configuraron los parámetros necesarios, se integró la función que comprende el bucle para el reporte de métricas. Esta función se conoce con el nombre de *poll_cycle*. La frecuencia con la que se ejecuta es de un minuto por lo que fue un requisito inicial el mantener la duración total de las tareas ejecutadas en su interior por debajo de esa cifra.

A nivel de código el poll_cycle base se define de la siguiente manera:

```
def poll_cycle
      report_metric 'NOMBRE_MÉTRICA', 'Value', VALOR_MÉTRICA
end
```

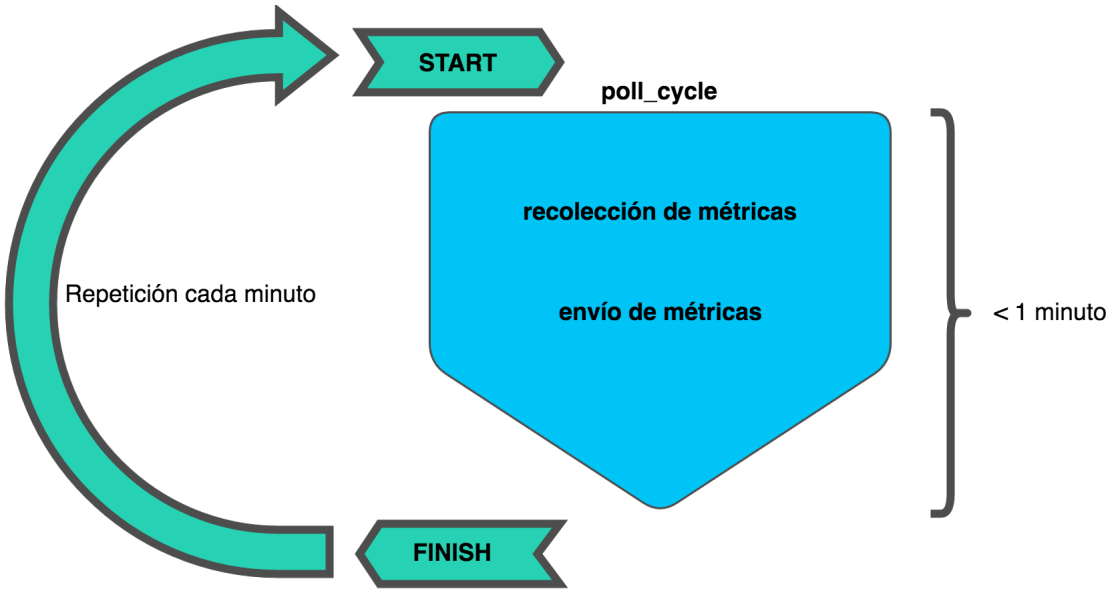

Figura 4-3 Diagrama de poll_cycle

Cada llamada a *report_metric* tiene como objetivo enviar una métrica en formato clave-valor. La clave se corresponde con el nombre de la métrica. Esta se ha diseñado siempre pensando en cómo se pretende agrupar métricas en las distintas gráficas. En los siguientes subapartados se argumentará la elección de los nombres de las métricas para cada servicio o recurso. Respecto al valor, NewRelic pone como limitaciones que este debe ser un número entero o decimal, en ningún caso podrá estar compuesto por cadenas de caracteres como nombres u otros tipos de valores.

Tras definir el *poll_cycle*, la última parte necesaria en la estructura es la instalación y la ejecución del agente. Esta se realiza con la llamada a dos funciones:

```
NewRelic::Plugin::Setup.install_agent :redborder, RedborderAgent
NewRelic::Plugin::Run.setup_and_run
```

Con la llamada a *install_agent* le pasamos como parámetros el objeto en formato YAML[10] que se corresponde con el nombre del agente dentro del fichero de configuración *newrelic_plugin.yml*. También le indicamos el nombre del módulo que compone el agente como segundo parámetro.

Por último se ejecuta la función que lanza el agente para empezar a reportar las métricas. Esta función es *setup_and_run*.

4.2 Agente: Recursos y servicios monitorizados

El propósito de este apartado es entrar en profundidad en el desarrollo y técnicas algorítmicas empleadas en el agente de manera específica para cada servicio o recurso que se monitoriza. Para cada uno se dedicará un subapartado específico con el fin de introducir las métricas extraídas y detallar la obtención y el procesamiento

[10] YAML es un formato de serialización de datos. Las siglas se corresponden con el inglés "Yet Another Markup Language".

de las mismas.

4.2.1 CPU

Con el fin de monitorizar la CPU se ha extraído una única métrica que permite obtener el porcentaje de CPU usada. Se entiende por CPU usada el tiempo que requiere la CPU para realizar una serie de tareas o ejecutar instrucciones.

Se trata de uno de los valores medibles clásicos de la monitorización. Permite conocer si el sistema está dimensionado correctamente a nivel de procesamiento en base a una utilización media o incluso valores de pico. Estos aspectos son bastante interesantes de detectar en base a la toma de decisiones de una arquitectura escalable.

4.2.1.1 Desarrollo

Para abordar la obtención de esta métrica se ha recurrido al protocolo SNMP. Los manager de Redborder tienen instalados un agente que permite la obtención de determinadas métricas a través de este protocolo. En este caso se ha accedido a la métrica recogida con el identificador (OID[11]) 1.3.6.1.4.1.2021.11.11.0. Esta se corresponde con el porcentaje de tiempo de CPU libre.

Como el objetivo es obtener el porcentaje de uso de CPU, se ha operado con el valor devuelto al obtener el objeto correspondiente al identificador anteriormente mencionado.

```
def cpu(manager)
  response = manager.get(['1.3.6.1.4.1.2021.11.11.0'])
  response.each_varbind do |vb|
    unless vb.nil?
      return 100 - vb.value.to_f
    end
  end
end
```

El código anterior muestra cómo se obtiene la métrica del porcentaje de uso de CPU. Se trata de una función que devuelve esta métrica a partir de una operación GET sobre el OID dado.

El parámetro que recibe es el manager creado para realizar las consultas SNMP. Es importante aclarar que este manager es una utilidad que nos permite realizar una serie de operaciones SNMP. Aunque comparta nombre con la máquina o instancia principal de un escenario con servicios de Redborder no tiene nada que ver.

Este manager SNMP se obtiene de la invocación necesaria a la apertura de la conexión que permite hacer las consultas:

```
SNMP::Manager.open(community: snmp_community, host: snmp_host) do
|manager|
```

Los parámetros que acepta son la comunidad SNMP y el host al que hacer las consultas. Estos parámetros son configurados en el fichero `newrelic_plugin.yml`. Esta estructura es el comienzo de un bucle para poder realizar otras consultas para otras métricas como se verá a continuación.

4.2.2 Memoria

De manera análoga al procedimiento para la métrica del uso en porcentaje de la CPU, se ha obtenido la

[11] Un OID (del inglés Object Identificator) tiene como fin identificar un determinado valor dentro de un grupo organizado jerárquicamente. Esta organización se realiza dentro de una MIB (del inglés Management Information Base).

métrica del porcentaje de uso de memoria. En este caso ha sido algo más complejo pues el procesado de los datos obtenidos ha requerido un número mayor de operaciones.

Con el objetivo obtener el porcentaje de uso de memoria y enviarlo a New Relic se ha necesitado obtener otras para poder calcular el objetivo a partir de una operación con estas.

4.2.2.1 Desarrollo

Para la obtención de las métricas con las que se opera también se ha utilizado SNMP y realizado las consultas a partir del mismo manager en el bucle de la conexión abierta.

A continuación se explicará el código utilizado para tales consultas.

```
def mem_total(manager)
  response = manager.get(['1.3.6.1.4.1.2021.4.5.0'])
  response.each_varbind do |vb|
    unless vb.nil?
      # puts "#{vb.name.to_s}
      #{vb.value.to_s}  #{vb.value.asn1_type}" unless vb.nil?
      return vb.value.to_f
    end
  end
end
```

Mediante la función anterior se obtiene el total de memoria RAM disponible en el equipo o instancia. Este es necesario para poder restarle otros valores relativos a la memoria con el fin de obtener la cantidad de memoria usada. Además también se usa el valor de la memoria total disponible para poder obtener un porcentaje.

```
def mem_free(manager)
  response = manager.get(['1.3.6.1.4.1.2021.4.6.0'])
  response.each_varbind do |vd|
     # puts "#{vd.name.to_s}  #{vd.value.to_s}  #{vd.value.asn1_type}"
     unless vd.nil?
    return vd.value.to_i
  end
end
```

Del mismo modo se obtiene mediante otra consulta el valor de la memoria *buffer* y *cache*. Estos valores nos indican qué parte de la memoria está siendo utilizada cachear bloques de disco en el caso de la primera, o información relativa a la lectura de ficheros en el en el caso de la segunda. Esta memoria puede ser liberada si algún proceso lo requiere pero en caso contrario agiliza la ejecución de tareas teniendo en cuenta que las operaciones I/O sobre el disco son lentas en comparación con otras.

```
def mem_total_buffer(manager)
  response = manager.get(["1.3.6.1.4.1.2021.4.14.0"])
  response.each_varbind do |vd|
     # puts "#{vd.name.to_s}  #{vd.value.to_s}  #{vd.value.asn1_type}"
     unless vd.nil?
    return vd.value.to_i
  end
end

def mem_total_cache(manager)
  response = manager.get(["1.3.6.1.4.1.2021.4.15.0"])
  response.each_varbind do |vd|
```

```
    # puts "#{vd.name.to_s}  #{vd.value.to_s}  #{vd.value.asn1_type}"
unless vd.nil?
    return vd.value.to_i
  end
end
```

Una vez obtenidos los valores anteriores se porcede a operar para obtener la métrica objetivo. Esta se corresponde con la siguiente ecuación:

$$\text{memoria usada (\%)} = \frac{(mem_{total} - mem_{free} - mem_{buffer} - mem_{cached})}{mem_{total}} * 100$$

El valor obtenido con la ecuación anterior es el que se visualiza en el dashboard para la métrica del porcentaje de memoria utilizada. No obstante se reportan el resto de valores a New Relic por si el usuario del plugin quisiera visualizar alguno de ellos con un objetivo más específico.

4.2.3 Prueba de red: Paquetes recibidos

Otra de las métricas reportadas es el porcentaje de paquetes recibidos. Nos permite conocer si existe algún problema en la configuración de red para alcanzar al equipo o instancia a través de su interfaz *bond0*. Esta es es una interfaz de tipo "bonding" que permite agregar otras interfaces de red. Es la que posee la dirección IP visible al resto del cluster y a la red privada de Redborder.

4.2.3.1 Desarrollo

Para enfocar la obtención de esta métrica se ha recurrido a la técnica utilizada por el servicio de monitorización propio de Redborder. Esta técnica consiste en el uso del comando `fping` que se ejecuta con el fin de enviar mensajes ICMP contra la interfaz anteriormente mencionada. El código usado para implementarlo es el siguiente:

```
def pkts_rcv(host)
  cmd = 'sudo /bin/nice -n 19 /usr/sbin/fping -p 1 -c 10 ' + host + '
2>&1 | tail -n 1 | awk \'{print $5}\' | sed \'s/%.*$//\' | tr \'/\' \'
\' | awk \'{print $3}\''
  return 100 - `#{cmd}`.strip.to_i
end
```

Como se puede comprobar se requiere un procesado importante a través de tuberías para extraer la información necesaria de la salida del comando `fping`. A continuación se muestra paso a paso cómo se procesa dicha información.

```
[[root@managerfjrios ~]# sudo /usr/sbin/fping -p 1 -c 10 managerfjrios 2>&1
managerfjrios : [0], 84 bytes, 0.03 ms (0.03 avg, 0% loss)
managerfjrios : [1], 84 bytes, 0.04 ms (0.03 avg, 0% loss)
managerfjrios : [2], 84 bytes, 0.04 ms (0.03 avg, 0% loss)
managerfjrios : [3], 84 bytes, 0.04 ms (0.03 avg, 0% loss)
managerfjrios : [4], 84 bytes, 0.03 ms (0.03 avg, 0% loss)
managerfjrios : [5], 84 bytes, 0.04 ms (0.03 avg, 0% loss)
managerfjrios : [6], 84 bytes, 0.04 ms (0.03 avg, 0% loss)
managerfjrios : [7], 84 bytes, 0.04 ms (0.03 avg, 0% loss)
managerfjrios : [8], 84 bytes, 0.03 ms (0.03 avg, 0% loss)
managerfjrios : [9], 84 bytes, 0.04 ms (0.03 avg, 0% loss)

managerfjrios : xmt/rcv/%loss = 10/10/0%, min/avg/max = 0.03/0.03/0.04
```

Figura 4-4 Salida del comando fping

En la figura Figura 4-4 Salida del comando fping podemos ver el resultado de la ejecución del comando

`fping` con las opciones `-p 1` y `-c 10` sobre el host `managerfjrios` (que se traduce en la IP de la interfaz bond0). La opción `-p` indica el número de milisegundos (uno en este caso) que se va a esperar a la recepción de la respuesta. Por otro lado la opción `-c` indica el número de paquetes ICMP que se enviarán al host contra el que se ejecuta el comando (en este caso diez). La salida de errores se redirecciona a la salida estándar haciendo uso del `2<&1`.

Esta salida debe ser procesada para poder obtener la información de los paquetes perdidos que está presente en la última línea de la captura anterior. A continuación se muestra una figura que muestra el procesado de la salida haciendo uso de tuberías y comandos como `tail`, `sed`, `awk` y `tr`.

```
[root@managerfjrios ~]# sudo /usr/sbin/fping -p 1 -c 10 managerfjrios 2>&1 | tail -n 1
managerfjrios : xmt/rcv/%loss = 10/10/0%, min/avg/max = 0.02/0.02/0.03
[root@managerfjrios ~]# sudo /usr/sbin/fping -p 1 -c 10 managerfjrios 2>&1 | tail -n 1
 | awk '{print $5}'
10/10/0%,
[root@managerfjrios ~]# sudo /usr/sbin/fping -p 1 -c 10 managerfjrios 2>&1 | tail -n 1
 | awk '{print $5}' | sed 's/%.*$//'
10/10/0
[root@managerfjrios ~]# sudo /usr/sbin/fping -p 1 -c 10 managerfjrios 2>&1 | tail -n 1
 | awk '{print $5}' | sed 's/%.*$//' | tr '/' ' '
10 10 0
[root@managerfjrios ~]# sudo /usr/sbin/fping -p 1 -c 10 managerfjrios 2>&1 | tail -n 1
 | awk '{print $5}' | sed 's/%.*$//' | tr '/' ' ' | awk {'print $3'}
0
```

Figura 4-5 Procesado de la salida de fping

Como se puede comprobar en la última línea de la figura anterior, se consigue procesar la salida del comando inicial para obtener el valor deseado.

A nivel de código en la función que tiene como fin devolver el valor de los paquetes enviados correctamente es necesario restarle el valor devuelto por el comando anterior a 100. De esta manera se obtiene la métrica que se desea reportar.

Por último es importante mencionar que al comando ejecutado en la función mencionada, le antecede la sentencia `/bin/nice -n 19`. Esta tiene como objetivo ejecutar el comando `fping` con la menor prioridad posible dado que `nice` es un comando que permite gestionar la prioridad con la que se ejecutan determinadas tareas. Las prioridades van desde -20 a 19 en orden de más prioridad a menos. Haciendo esto conseguimos que la ejecución de `fping` no intervenga en otras tareas más importantes.

4.2.4 Prueba de red: Latencia

En este caso la métrica reportada a explicar es la correspondiente a la latencia. La latencia se define como el tiempo transcurrido desde que un origen emite un paquete y el destino lo recibe. En cierto modo su obtención y procesado es análogo al de los paquetes recibidos correctamente.

4.2.4.1 Desarrollo

El comando `fping` vuelve a ser de nuevo utilizado para la obtención de la métrica de la latencia. En este caso su uso es distinto tanto a nivel de opciones como a nivel de procesado de la salida. El código correspondiente a la función que obtiene la latencia es el siguiente:

```
def latency(host)
  cmd = 'sudo /bin/nice -n 19 /usr/sbin/fping -q -s ' + host + ' 2>&1|
grep \'avg round trip time\'| awk \'{print $1}\''
  return `#{cmd}`.strip.to_i
end
```

El comando ejecutado de nuevo hace uso de `nice` para darle prioridad 19 (la mínima prioridad posible) al proceso de `fping`. Este en este caso hace uso de las opciones `-q` y `-s`. La opción –q se utiliza para que el comando `fping` no devuelva resultados mientras envía los paquetes. Del mismo modo la opción –s tiene como objetivo mostrar las estadísticas acumuladas al terminar de enviar todos los paquetes.

De nuevo se procede como en el caso anterior, se mostrará a continuación el resultado de la ejecución del comando `fping` con las opciones pertinentes:

```
[[root@managerfjrios ~]# fping -q -s managerfjrios

       1 targets
       1 alive
       0 unreachable
       0 unknown addresses

       0 timeouts (waiting for response)
       1 ICMP Echos sent
       1 ICMP Echo Replies received
       0 other ICMP received

 0.05 ms (min round trip time)
 0.05 ms (avg round trip time)
 0.05 ms (max round trip time)
        0.000 sec (elapsed real time)
```

Figura 4-6 Salida de fping -q -s HOST

El procesado de esta información se vuelve a hacer con tuberías y comandos como `grep` y `awk`:

```
[[root@managerfjrios ~]# fping -q -s managerfjrios 2>&1 | grep 'avg round trip time'
 0.03 ms (avg round trip time)
[root@managerfjrios ~]# fping -q -s managerfjrios 2>&1 | grep 'avg round trip time' | awk {'print $1'}
0.03
```

Figura 4-7 Procesado de fping para la latencia

Realmente lo que se está obteniendo es el Round Trip Time (o RTT). Este no es más que el tiempo que tarda en llegar la respuesta de un determinado paquete desde una fuente a un destino. Al ser la respuesta y no la solicitud se está midiendo el tiempo de ida de la solicitud y el de vuelta de la respuesta. Idealmente la latencia se calcularía dividiendo el valor del RTT por dos pero es bastante común encontrar en algunos SLAs[12] el término latencia como RTT.

4.2.5 AVIO

El Avio puede resultar una de las métricas menos conocidas de todas las mencionadas hasta ahora. Se trata de un valor relacionado con el estado del disco y básicamente mide el número de milisegundos que se necesitan sobre el mismo.

4.2.5.1 Desarrollo

A nivel práctico, la obtención del Avio se ha enfocado como otra función que devuelve el valor a reportar con la métrica asociada:

```
def get_avio()
  return `(atop 2 2 | grep avio |  awk '{print $15}' |
  paste -s -d'+' | sed 's/^/scale=3; (/' | sed 's|$|)/2|' | bc)`
end
```

Para la obtención del valor objetivo se ha recurrido al comando de bash `atop`. Este se define en su manual como una utilidad avanzada para monitorizar el sistema y los procesos. Se omite una captura de salida con el comando sin el procesamiento dado que ocuparía toda la pantalla y únicamente nos interesa el posterior procesado para obtener el avio.

El comando `atop` se ejecuta junto con dos parámetros (`atop 2 2`). El primero de ellos se refiere al

[12] Un SLA (del inglés Service Level Agreement) es un acuerdo a nivel de servicios que una empresa contrae con un cliente al realizar la venta de un servicio.

intervalo (en segundos) de refresco de los valores monitorizados, es decir, con qué frecuencia va a obtener los valores a mostrar. El segundo de ellos corresponde al número de muestras que se necesitan. La duración de la pantalla interactiva del comando será el resultado de multiplicar el primer y el segundo parámetro, en nuestro caso 4 segundos.

4.2.6 Disco utilizado

Otra de las métricas obtenidas es el porcentaje de espacio utilizado en el disco. Del mismo modo que la CPU o la memoria, el porcentaje de disco que se ha utilizado se trata de un valor que permite valorar y tomar decisiones acerca del dimensionamiento de la máquina o instancia de la que se obtiene.

4.2.6.1 Desarrollo

Análogamente al procedimiento de los recursos de Memoria y CPU se ha hecho uso de SNMP para obtener esta métrica. De hecho la estructura de la función es la misma y únicamente cambia el OID utilizado específicamente para esta métrica:

```
def disk_percent(manager)
  response = manager.get(["1.3.6.1.4.1.2021.9.1.9.1"])
  response.each_varbind do |vd|
    # puts "#{vd.name.to_s}  #{vd.value.to_s}  #{vd.value.asn1_type}"
unless vd.nil?
    return vd.value.to_i
  end
end
```

El valor devuelto es directamente el que se obtiene de la consulta convertido a entero. Este es el valor final que se reportará a New Relic.

4.2.7 Carga de disco

Se entiende como carga de disco la cantidad de operaciones I/O realizadas de media en períodos de un minuto tal y como establece la descripción de la tabla consultada por SNMP. Como se podrá comprobar ahora, para poder obtener esta métrica es necesario hacer una consulta más completa que las vistas hasta ahora. Por ello se va a hacer a través del comando de bash y se procesará la información con tuberías y otros comandos.

4.2.7.1 Desarrollo

Para tener una referencia se muestra la tabla desde la que se obtiene el dato objetivo:

```
[[root@managerfjrios ~]# snmptable -v 2c -c redBorder 127.0.0.1 diskIOTable
SNMP table: UCD-DISKIO-MIB::diskIOTable

 diskIOIndex diskIODevice diskIONRead diskIONWritten diskIOReads diskIOWrites diskIOLA1
           1         ram0          0              0          0           0         0
           2         ram1          0              0          0           0         0
           3         ram2          0              0          0           0         0
          28         sda2   78554112     1726980096     190363     3351984         0
          29         dm-0   78205952     1726980096     200756    33976058         0
```

Figura 4-8 Consulta DiskIOTable

Para poder entender cómo se obtiene la métrica en este caso es necesario introducir el concepto de LVM. Este es un administrador de volúmenes lógicos para el kernel de Linux. Gracias a LVM se puede tener una vista de alto nivel sobre el almacenamiento de sistemas Linux. En la Figura 4-8 Consulta DiskIOTable de puede ver que en la última línea aparece un dispositivo con el nombre de `dm-0`. Este es una de las partes del mapeo de dispositivos de LVM. Podríamos decir que actúa como una capa de abstracción sobre los volúmenes lógicos.

Es de la línea correspondiente al dispositivo `dm-0` de donde se obtiene el valor que se va a reportar como se puede ver en el código encargado de ello:

```
def disk_load
  return `(snmptable -v 2c -c redBorder 127.0.0.1 diskIOTable | grep '
dm-0 ' | awk '{print $7}')`.strip.to_i
end
```

4.2.8 Memoria por servicios

Además de la métrica del porcentaje de uso de memoria reportada a New Relic, también se envían métricas del porcentaje de uso de memoria por servicio. Los servicios sobre los que se reporta esa métrica son:

- Kafka
- Nprobe
- PostgreSQL
- RBWebUI
- Zookeper
- Druid: Dividiendo a su vez por tipos de nodos (Realtime, Coordinator, Historical, Broker)

El objetivo de enviar las métricas relativas al consumo de memoria de estos servicios en porcentaje se fundamenta principalmente en la visualización en una sola gráfica de los recursos consumidos por los mismos. Los servicios que se han seleccionado se corresponden con los principales que componen la estructura de una configuración básica del manager de Redborder así como el servicio encargado de gestionar la interfaz web.

4.2.8.1 Desarrollo

Para la obtención de estas métricas se ha hecho uso del script alojado en la ruta `/opt/rb/bin/rb_mem.sh`. A este script se le pasa como parámetro con la opción `-f` un fichero que contenga el PID del servicio. El script devolverá la memoria ocupada por el servicio al que corresponda el fichero proporcionado. Esta memoria se divide entre el total disponible en el sistema y se obtiene el porcentaje de memoria ocupada por servicio.

El contenido del fichero es bastante sencillo y, aunque no haya sido desarrollado como parte de este trabajo se explicará a continuación:

```
#!/bin/bash

function usage(){
  echo "$0 [ -p pid ] [ -f <filepid> ]"
}

pid=""
file=""

while getopts "p:f:h" opt; do
  case $opt in
    p) pid=$OPTARG;;
    f) file=$OPTARG;;
    h) usage;;
  esac
done

if [ "x$file" != "x" ]; then
  pid=$(head -n 1 $file)
fi

if [ "x$pid" != "x" ]; then
```

```
  if [ -f /proc/$pid/cmdline ]; then
    pmap -x $pid | grep total | awk '{print $4}' | sed 's/K//'
  fi
fi
```

Con el código expuesto podemos analizar paso a paso las tareas que realiza el script usado para obtener el porcentaje de memoria ocupada por un servicio determinado. La primera función tiene como objetivo mostrar las opciones disponibles en el caso de que se solicite con la opción `-h`. El siguiente bucle `while` almacena el valor de los argumentos en el caso de que la opción `-f` o `-p` sea seleccionada. Por último se obtiene el PID ya sea a partir del fichero proporcionado con la opción `-f` o bien por el argumento en caso de que este PID se pase directamente. El comando utilizado para obtener la memoria a partir del PID es `pmap`.

El comando `pmap` se llama con la opción `-x` y se le pasa el PID. De la salida únicamente interesa la línea que contiene los valores totales pues es en esta donde se toma la cuarta columna que contiene el valor deseado.

El valor devuelto por el script anterior se procesa para ponderarlo en base a la memoria total y convertirlo a porcentaje multiplicando por 100. Estas operaciones se realizan en el fichero principal (`newrelic_redborder_agent`) justo al determinar el valor a reportar en cada memoria asociada a un servicio. A continuación se adjunta el código relativo a las funciones que realizan la obtención de estos valores:

Memoria de los nodos broker de Druid

```
def memory_total_druid_broker
  return `(sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/druid_broker/supervise/pid 2>/dev/null)`.strip.to_i
end
```

Memoria de los nodos coordinator de Druid

```
def memory_total_druid_coordinator
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/druid_coordinator/supervise/pid 2>/dev/null`.strip.to_i
end
```

Memoria de los nodos historical de Druid

```
def memory_total_druid_historical
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/druid_historical/supervise/pid 2>/dev/null`.strip.to_i
end
```

Memoria de los nodos realtime de Druid

```
def memory_total_druid_realtime
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/druid_realtime/supervise/pid 2>/dev/null`.strip.to_i
end
```

Memoria de Kafka

```
def memory_total_kafka
  return `sudo /opt/rb/bin/rb_mem.sh -f
```

```
/opt/rb/var/sv/kafka/supervise/pid 2>/dev/null`.strip.to_i
end
```

Memoria de Nprobe

```
def memory_total_nprobe
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/nprobe/supervise/pid 2>/dev/null`.strip.to_i
end
```

Memoria de PostgreSQL

```
def memory_total_postgresql
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/postgresql/supervise/pid 2>/dev/null`.strip.to_i
end
```

Memoria del servicio RB-WebUI

```
def memory_total_rbwebui
  return `sudo /opt/rb/bin/rb_mem.sh -f /opt/rb/var/sv/rb-
webui/supervise/pid 2>/dev/null`.strip.to_i
end
```

Memoria de Zookeeper

```
def memory_total_zookeeper
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/zookeeper/supervise/pid 2>/dev/null`.strip.to_i
end
```

Como se ha comentado en líneas anteriores, estas funciones son llamadas directamente en las sentencias `report_metric` pertinentes donde se opera para obtenerlo en formato de porcentaje y reportarlo directamente de ese modo.

4.2.9 Druid

Como se introdujo en el apartado 3.6 del anterior capítulo, Druid es uno de los principales servicios del núcleo de un manager de Redborder. Esta es una de las razones por las que se le presenta un trato personalizado en la sección de dashboards como se explicará en el apartado 4.4. Otra de las razones por las que Druid cobra especial importancia es por la ausencia de plugin específico para el servicio. Sin embargo de Kafka y otros principales encontramos plugins publicados en la plataforma de New Relic.

En este subapartado vamos a explicar todos los tipos de métricas que posee Druid así como su extracción, procesado y posterior preparación para el envío. Incidiremos en aquellas en las que se ha mostrado más interés por parte de los expertos de Druid de Redborder dado que son las que se van a seleccionar para mostrar gráficas en la plataforma de New Relic. No obstante es importante dejar claro que el agente reporta todas las métricas existentes de Druid de manera dinámica. En este punto se ha optimizado la eficiencia tanto a nivel de precisión de los valores reportados como a nivel de tiempo necesario para obtener y procesar las métricas. Estos aspectos se tratarán con mayor profundidad en el subapartado 4.2.9.1 de Desarrollo.

Principalmente las métricas generadas por Druid se enfocan en las consultas, ingestión de datos y coordinación del servicio. Además existen otras relativas al consumo de recursos y otros parámetros útiles de los nodos de

Druid.

Las métricas de Druid se almacenan en objetos JSON que son almacenados en ficheros log en tiempo de ejecución o enviado vía http a servicios enfocados a colas de mensajes como Kafka. Ante una instalación genérica de Druid es necesario activar la generación de métricas puesto que por defecto esta función está deshabilitada.

Las métricas tienen un formato común que se presenta a continuación en la Figura 4-9 Formato de una métrica: Objeto JSON.

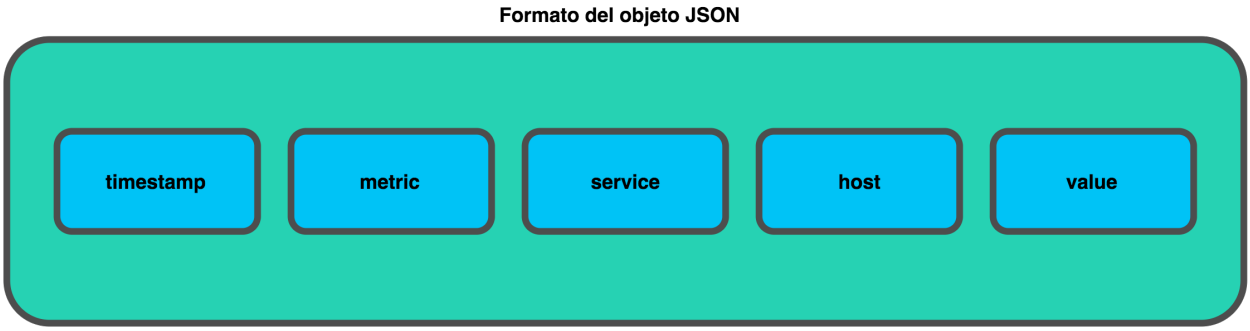

Figura 4-9 Formato de una métrica: Objeto JSON

De todos los campos que posee una métrica de Druid únicamente nos interesaremos por los campos metric, service y value. No obstante se definen todos:

- **Timestamp:** Se trata de un campo para identificar el momento en el que se ha generado la métrica.
- **Metric:** Se corresponde con el nombre de la métrica.
- **Service:** Es el nombre del servicio o rol de druid al que corresponde la métrica en cuestión.
- **Host:** Su objetivo es almacenar el nombre del host que ha generado la métrica.
- **Value:** Siempre será un valor numérico asociado con la métrica.

Antes de entrar en aspectos relativos al desarrollo se explicará las distintas métricas existentes en Druid []. Estas son de distinto tipo y las clasificaremos en base a su ámbito. Con este criterio encontramos los siguientes grupos:

- **Query Metrics:** Abarca todas las métricas relativas a las consultas.
- **Ingestion Metrics**: Estas métricas solo están disponibles si en la configuración de los nodos Realtime está incluido el RealtimeMetricsMonitor como es el caso del manager de Redborder. Principalmente comprende métricas de los periodos de emisión.
- **Coordination**: Estas métricas son específicas de los nodos Coordinator y se resetean cada vez que estos nodos ejecutan la lógica de coordinación.
- **General Health**: Las métricas de este tipo permiten comprobar que no existen problemas generales en la ejecución de uno o varios tipos de nodos en Druid. Por ejemplo métricas relativas a la Java Virtual Machine.
- **Sys**: Este último grupo contiene métricas del consumo de CPU, memoria o uso del disco de los nodos de Druid. No se reportarán en ningún caso métricas de este tipo dado que el manager de Redborder no contiene el módulo SysMonitor y esta es una condición indispensable para obtener estos valores.

Explicaremos a continuación una a una las métricas que se van a visualizar dentro del dashboard de Druid en el plugin:

Métrica	Tipo	Descripción
query/time	Query Metrics	Milisegundos necesarios para completar una consulta.
jvm/mem/used	General Health	Memoria usada.
jvm/mem/init	General Health	Memoria inicial.
jvm/mem/committed	General Health	Memoria comprometida. (reservada)
jvm/mem/max	General Health	Memoria máxima disponible.
ingest/events/processed	Ingestion Metrics	Número de eventos procesados correctamente.
ingest/events/thrownAway	Ingestion Metrics	Número de eventos rechazados porque se encontraban fuera del período de recepción.
ingest/persist/time	Ingestion Metrics	Número de milisegundos haciendo "intermediate persist".[13]
ingest/merge/time	Ingestion Metrics	Número de milisegundos transcurridos haciendo agregación de segmentos de tipo "intermediate".
segment/count	Coordination Metrics	Número de segmentos disponbles.

4.2.9.1 Desarrollo

Una vez comentadas las distintas métricas que se van a visualizar en el dashboard de Druid (llamado "rb-druid") del plugin de Redborder se va a explicar cómo se ha desarrollado la obtención y procesado de todas las métricas existentes. Esto hace posible la existencia de un carácter dinámico en la adaptación del dashboard a las necesidades del usuario final. Si este quiere visualizar otra métrica concreta la tendrá disponible aunque no se haya configurado un dashboard para visualizarla.

La primera aproximación que se desarrolló para la obtención de estas métricas se basó en la técnica del "hard-coding". Esta no tuvo otro objetivo que validar la visualización de determinadas métricas. Cuando esta se produjo se empezó a diseñar un algoritmo que fuera más escalable.

Este algoritmo permitía que se almacenasen las métricas de manera que se fueran actualizando si se recibían valores nuevos de una métrica ya almacenada. En un principio se tomó como solución final a falta de las pruebas. Es en la ejecución de estas cuando se detectó el problema de que las métricas no se añaden a los logs de druid de forma periódica sino que si existe un valor pico muy puntual, se transmite el valor pico de manera inmediata. Del mismo modo cuando se vuelve a normalizar la métrica se sobreescribe el valor de pico con el valor normal. En el apartado de validación y pruebas del capítulo 5 se explicará el proceso mediante el que se encontró estas irregularidades y que provocó la corrección de la implementación. En este subapartado se va a abordar con gran detalle cuál ha sido la implementación final.

[13] El concepto de "Intermediate Persist" en Druid se refiere al proceso agrupación de los datos en segmentos, su compresión y envío a S3. Para evitar pérdidas importantes causadas por algún fallo, esta agrupación se hace cada cierto tiempo dentro del ciclo completo (de ahí el término "intermediate").

Como punto inicial de la explicación de la solución final es importante mencionar que las métricas se obtienen directamente de los logs de Druid. Estos logs se leen de manera dinámica, es decir, dependiendo de una configuración u otra habrá más o menos logs. Lo que sí es estático es el directorio donde se encontrarán estos logs, este será el mismo siempre (`/var/log/druid`) en base a la configuración de Redborder. Además también se añaden métricas recogidas desde otros ficheros log que se incluyeron más tarde a petición del equipo experto en Druid de Redborder. Estos logs adicionales se encuentran en la ruta `/tmp/druid-indexing/persistent/tasks/index_*/log`.

A nivel de código la función principal que comienza con el proceso de obtención de métricas de druid se llama en el fichero newrelic_redborder_agent. Esta función se llama druid_master y su objetivo es abrir un hilo a partir del proceso principal para leer los valores que se van añadiendo a los logs. El código de esta función es el siguiente:

```
def druid_master
  threads = []
  Dir.glob('/var/log/druid/*.log') do |log_file|
    # puts "Started At #{Time.now} with file: " + log_file
    $logger.debug("Started metrics parser with file: " + log_file)
    t = Thread.new do
      log_handler_druid(log_file)
    end
    threads << t
  end

  Dir.glob('/tmp/druid-indexing/persistent/tasks/index_*/log') do
  |log_file|
    # puts "Started At #{Time.now} with file: " + log_file
    $logger.debug("Started metrics parser with file: " + log_file)
    t = Thread.new do
      log_handler_druid(log_file)
    end
    threads << t
  end
end
```

Como se puede comprobar, cada hilo procesa un fichero log a través de la función `log_handler_druid` a la que se le pasa por parámetro el fichero adecuado. Esta función se encuentra en el fichero `tail_f_druid.rb` cuyo código se incluye con una sentencia `require_relative`. El código de esta función se expone a continuación:

```
def log_handler_druid(filename)
  File.open(filename, 'r') do |log|
    log.extend(File::Tail)
    log.backward(1)
    log.tail { |line| druid_parser(line, filename) }
  end
end
```

Esta función hace uso de las funciones incluidas con la gema[14] `file-tail`. Esta parte del código es la que nos permite que cada línea nueva añadida al fichero de log que proceda se pueda procesar para extraer la

[14] En ruby una gema es un conjunto de funciones y recursos encapsulados que permiten añadir funcionalidades más concretas que enriquezcan los programas. Es un copceto análogo al de otros entornos como Node.js que hace uso de los módulos NPM.

métrica objetivo en el caso de que esta exista. De ahí es donde viene su nombre ya que sería una función análoga al comando `tail -f` en línea de comandos Linux. Como se puede comprobar por cada línea nueva que se añade se llama a la función druid_parser. Esta es la función que implementa la parte más interesante del algoritmo diseñado para cumplir los requisitos de la manera óptima. Por esta razón, antes de mostrar el código se va a explicar esta parte del algoritmo de manera conceptual.

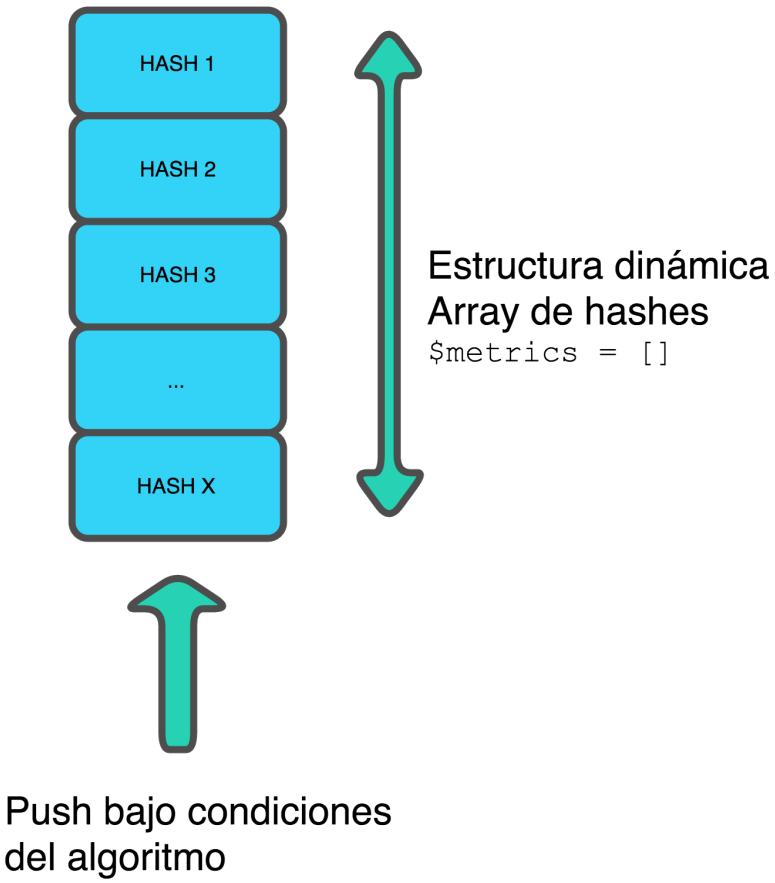

Figura 4-10 Array de hashes para Druid

Las métricas que se van obteniendo se van incluyendo en un array dinámico de hashes tras ser procesadas y cumplir determinadas condiciones. Estos hashes se interpretan como una unidad o bloque con datos relacionados con las distintas métricas.

A continuación se expondrá la estructura de datos que compone un hash para el array de métricas de Druid:

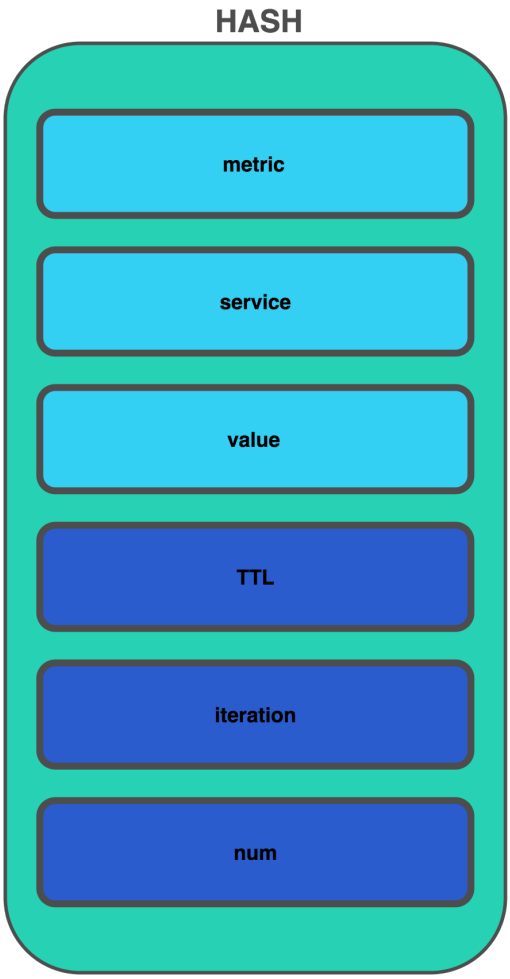

Figura 4-11 Estructura de datos de los hashes de Druid

Como se puede comprobar en la Figura 4-11 Estructura de datos de los hashes de Druid se tienen seis tipos de datos dentro de un hash para una métrica de Druid. Mediante el código de colores se diferencian dos grupos de datos. Los de azul claro serán datos que compondrán la información que se reporta a New Relic. Por el contrario, los de azul oscuro se usarán para gestionar las estructuras dinámicas dentro del agente. Cada campo tiene una función determinada por lo que se va a explicar qué contendrá cada uno así como su función:

- **Metric:** Contiene el nombre de la métrica. Permite identificar una métrica dentro del array.
- **Service:** Análogamente al campo *metric* contiene el servicio o rol de Druid que ha emitido la métrica. Es importante porque varios tipos de nodos de Druid pueden reportar la misma métrica.
- **Value:** Contiene el valor numérico de la métrica. Este será el valor final reportado a New Relic.
- **TTL:** Es un campo de gestión. Permite mantener el valor de una métrica durante un máximo de tres minutos si esta nos se actualiza antes. Este campo se incluyó para evitar tener gráficas irregulares sin reportar valores "fantasma". Además es importante para actualizar las métricas como se verá en el algoritmo de la Figura 4-12 Algoritmo de gestión de las métricas de Druid.
- **Iteration:** Este campo se utiliza para añadirle a una métrica el número de iteración del *poll_cycle* en el que se ha actualizado. Su fin principalmente es facilitar la tarea de depuración del código así como obtener información acerca de la frecuencia con la que se actualizan las métricas.
- **Num:** Es un valor que se incluyó en la última validación del agente. Su fin es poder mantener dos métricas iguales generadas por el mismo servicio pero con distinto valor. Anteriormente se ha comentado que ante cambios bruscos se podía actualizar el valor de una métrica cubriendo un valor de pico puntual. Mediante la adición de una segunda métrica se evita este problema. Puede surgir la

pregunta de por qué no se incluyen más réplicas de distinto valor de una misma métrica y servicio. La respuesta radica en las limitaciones de New Relic. Esta plataforma solo permite tener un máximo de dos valores para una misma métrica por minuto.

Habiendo explicado el array de hashes así como las estructuras de datos que componen cada uno nos encontramos en posición de abordar el algoritmo que determinará si la llegada de una métrica provocará la actualización de una ya existente en el array o la inclusión de una nueva. Este algoritmo se presenta en la Figura 4-12 Algoritmo de gestión de las métricas de Druid.

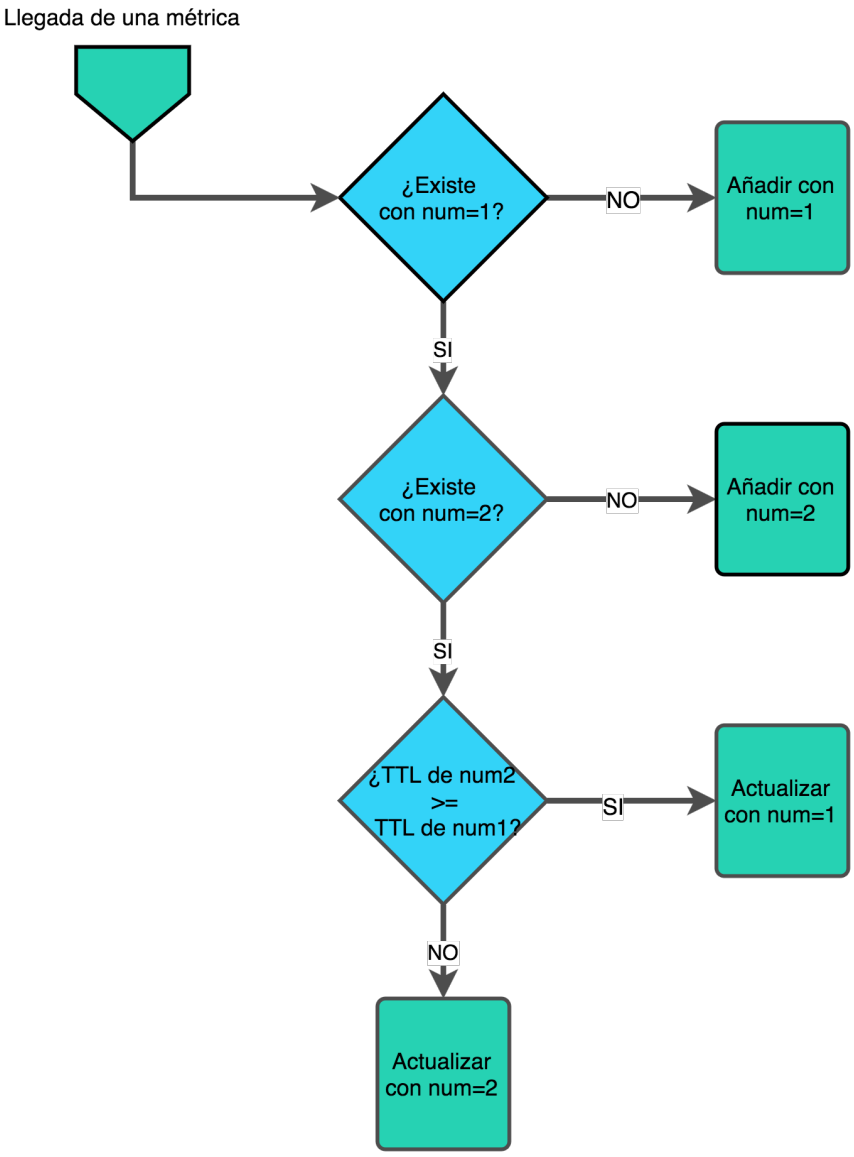

Figura 4-12 Algoritmo de gestión de las métricas de Druid

Este algoritmo entra en juego en la función `druid_parser`. Se trata de la última función en el procesado de las métricas de Druid. Para explicar las tareas que se realizan en ella se expondrá el código de la misma:

```
def druid_parser(line, file)

  unless line.nil?
    unless line.match(/"service":"(.*)","host/).nil? ||
```

```
line.match(/"metric":"(.*)","value/).nil? ||
line.match(/"value":(\d+),"/).nil?
      service = line.match(/"service":"(.*)","host/)[1]
      if service == 'middleManager'
        unless line.match(/"taskId":\["(.*)"\]}\]/).nil?
          service = line.match(/"taskId":\["(.*)"\]}\]/)[1]
        end
      end
      metric = (line.match(/"metric":"(.*)","value/)[1]).tr('/', '_')
      value = line.match(/"value":(\d+),"/)[1]
    end
    unless service.nil? || value.nil?
      found1 = false
      found2 = false
      ttl1 = 3
      ttl2 = 3
      $metrics.each do |m|
        if m["metric"] == metric && m["service"] == service &&
        m["num"] == 1
          found1 = true
          ttl1 = m["ttl"]
        end
      end
      if found1
        $metrics.each do |m|
          if m["metric"] == metric && m["service"] == service &&
           m["num"] == 2
            found2 = true
            ttl2 = m["ttl"]
          end
        end
      end
      if found1 && found2
        if ttl2 >= ttl1
          $metrics.each do |m|
            if m["metric"] == metric && m["service"] == service &&
            m["num"] == 1
              m["value"] = value
              m["ttl"] = 3
              m["iteration"] = $i
            end
          end
        else
          $metrics.each do |m|
            if m["metric"] == metric && m["service"] == service &&
            m["num"] == 2
              m["value"] = value
              m["ttl"] = 3
              m["iteration"] = $i
            end
          end
        end
      end
      if !found1 || !found2
        if !found1
          num = 1
        end
```

```
          if found1 && !found2
            num = 2
          end
          $metrics << {
            "metric" => metric,
            "service" => service,
            "value" => value,
            "ttl" => 3,
            "iteration" => $i,
            "num" => num
          }
        end
      end
    end
  end
```

El inicio de esta función se corresponde con el parseo con expresiones regulares de la línea que se va a procesar para extraer la métrica. Con ello se extrae la métrica, el servicio y el valor que se almacenarán en un hash del array de hashes. Cabe destacar que en este punto también se hace la distinción de un servicio llamado *middleManager* que consiste en un rol especial de Druid.

Tras extraer los datos objetivo (métrica, servicio y valor) se ejecuta el algoritmo que añadirá un nuevo hash al array en el caso de que no exista la métrica+servicio o exista una única vez. Si existe dos veces se actualizará la que tenga menor TTL.

En este punto hemos explicado la parte de la obtención y procesado de métricas de Druid. Los aspectos relativos al envío hacia la plataforma de New Relic se abordarán en el apartado Agente: Otros aspectos importantes.

4.2.10 Checks

Otra de las métricas importantes reportadas por el agente se corresponden con los checks de los servicios que se añadan en el fichero de configuración. Estas métricas pueden valer 1 si el servicio se está ejecutando ó 0 en el caso contrario. Los servicios sobre los que hará el check son aquellos que deberían estar ejecutándose en el manager en base a la configuración inducida por Chef.

4.2.10.1 Desarrollo

Para implementar la obtención de estas métricas se ha creado una función principal (`check_master`) que ante una llamada a la misma con un servicio pasado como parámetro añade o actualiza el hash correspondiente a un array de hashes creado específicamente para los checks. Este array se define con la siguiente sentencia:

```
$check = []
```

Las hashes que contendrá serán más simples que los ya vistos para Druid. Estos tendrán la siguiente estructura:

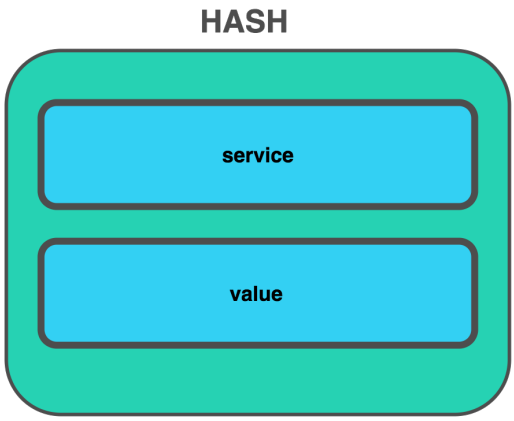

Figura 4-13 hash de los checks

Los campos con los que cuenta esta estructura son service y value. El primero de ellos incluirá el servicio sobre el que se ejecutará el check. El segundo de ellos, como se ha comentado en líneas anteriores, vale 1 en caso de que el servicio esté en ejecución y 0 si no lo está.

La gestión de estas estructuras de datos se hacen en la función `check_master`. El código de esta se adjunta a continuación:

```
def check_master(service)
    $logger.debug(service + ' service checked')

    cmd = 'lib/rb_nr_check ' + service
    value = `#{cmd}`
    status = ($?.to_s.split(' ')[3] == '1') ? 1 : 0
    puts 'service is ' + service + ' then status is ' + status.to_s


    found = false

    $check.each do |m|
      if m['service'] == service
        found = true
        m['value'] = status
      end
    end
    if !found
      $check << {
        'service' => service,
        'value' => status
      }
    end
end
```

Dentro de `check_master` se hace uso de un script desarrollado en bash y que se encuentra en el fichero `lib/rb_nr_check`. Este script se explicará en detalle en las siguientes líneas. Antes de ello es importante mencionar lo que hace `check_master` con el valor devuelto por el mismo. Este se procesa para almacenar un 0 ó un 1 según corresponda en la variable status.

Para concluir este subapartado se muestra el script que permite identificar si un servicio se encuentra en ejecución o no:

```
#!/bin/sh
```

```
SERVICE="$1"
RESULT=`ps aux | grep ${SERVICE} | grep -c -v grep`
if [ "${RESULT:-null}" -le "2" ]; then
    echo "${SERVICE} not running"
    RETVAL=0
else
    echo "running: ${SERVICE}"
    RETVAL=1
fi

exit "$RETVAL"
```

Se trata de un script bastante simple que recoge un servicio como primer argumento en su llamada. Este servicio se utilizará para hacer un `grep` a la salida de `ps aux`. Además se contarán las líneas de esta salida para determinar si es un proceso en ejecución o por el contrario no está en servicio.

Llegados a este punto solo queda comentar cómo se envían estas métricas pero esto se dejará para el apartado 4.3.

4.2.11 Chef

En el apartado 3.3 de este documento se comentó que Chef era un servicio que tenía como objetivo la correcta configuración de los manager a través del Chef Client. Dada la importancia de este proceso se decidió integrarlo en el agente del plugin. Tras estudiar con el equipo de sistemas de Redborder las posibles métricas que se podían reportar de Chef, se llegó a la conclusión de que lo más interesante era reportar los errores de Chef y poder visualizarlos de manera semidetallada en la plataforma.

4.2.11.1 Desarrollo

De nuevo se ha enfocado el reporte de las métricas conservando el carácter dinámico que ha predominado a lo largo del desarrollo del agente. La metodología se basa en la lectura de las líneas que se van añadiendo al fichero `/var/log/chef-client/current`.

Sobre estas líneas se buscarán los errores para almacenar el mensaje que interese en el caso de que dichos errores ocurran. Para poder explicar con precisión este proceso se va a exponer el código haciendo alusión a las funcionalidades implementadas.

La función principal se ha desarrollado de manera análoga a otras como la de Druid o la de los Check. Esta recibe el nombre de `chef_master`. Dicha función se llama en el fichero principal del agente, `newrleic_redborder_agent` y también se apoya en un array de hashes para almacenar las métricas, `$chef = []`.

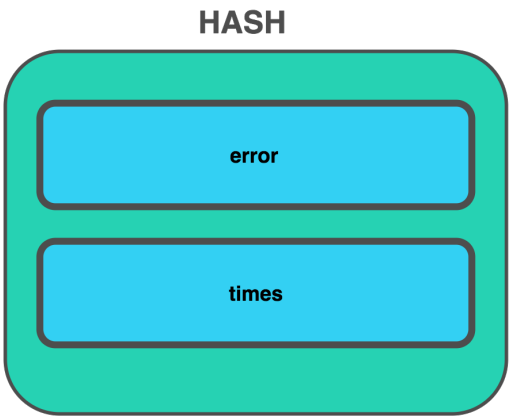

Figura 4-14 hash de Chef

El código de `chef_master` es el siguiente:

```
def chef_master
  threads = []
    log_file = '/var/log/chef-client/current'
    # puts "Started At #{Time.now} with file: " + log_file.to_s
    $logger.debug("Started chef error parser with file: " + log_file)
    t = Thread.new do
      log_handler_chef(log_file)
    end
    threads << t
end
```

Como se puede comprobar esta función abre un nuevo hilo para el manejo del fichero del que se van a obtener las métricas, De esta manera se evita consumir el tiempo del `poll_cycle` del proceso principal. Este hilo se crea para ejecutar la función log_handler_chef. El código de esta se expone a continuación:

```
def log_handler_chef(filename)
  File.open(filename, 'r') do |log|
    log.extend(File::Tail)
    log.backward(1)
    log.tail { |line| chef_parser(line) }
  end
end
```

Esta función recibe como argumento el fichero log del que se leerán los errores. Se trata del mismo proceso explicado en apartados anteriores para reproducir el funcionamiento de un `tail -f` en línea de comandos de Linux. En este caso la función que procesará cada línea añadida al fichero será `chef_parser`.

```
def chef_parser(line)
  unless line.nil?
    unless line.match(/ERROR:(.*)/).to_s.nil?
      matched = line.match(/ERROR:(.*)/).to_s
      error = (matched.include? 'retry') ? matched.split(',')[0] :
              matched
    end
    unless error.nil? || error.empty?
      found = false
      $chef.each do |m|
        if m["error"] == error
          found = true
          m["times"] = 1
        end
      end
      if !found
        $chef << {
          "error" => error,
          "times" => 1
        }
      end
    end
  end
end
```

Esta función sería la análoga a `druid_parser`. Como se puede comprobar a nivel de código es más sencilla

puesto que no es necesario un procesado tan completo como en el caso de Druid. Principalmente se parsean las líneas nuevas para comprobar en cada una si existe algún error haciendo uso de expresiones regulares. En ese caso se extrae el mensaje y se asigna a la variable `error` tras eliminar información innecesaria.

Tras procesar la información se gestiona el array para incluir un nuevo hash o actualizar uno ya existente. Desde este array se enviarán los datos a la plataforma como se verá en el apartado 4.3.

4.2.12 Nginx

El último servicio sobre el que se reporta es Nginx. Este servicio entre otras tareas sirve la interfaz web por lo que se considera también un punto crítico dentro del mánager de Redborder. También actúa de balanceador de carga e incluso de intermediario de Chef. De manera previa a realizar la inclusión de métricas relativas a este servicio se consultó con el equipo de web de Redborder para poder determinar qué aspectos sería interesante monitorizar. Tras dicha consulta se llegó a la conclusión de que lo más interesante sería contabilizar los códigos de estado que se devolvían en cada acceso a la web del manager.

Los códigos de estado se incluyen en las respuestas HTTP y en su mayoría están recogidos en RFCs como RFC2616, RFC2518, RFC2817, RFC2295, RFC2774 y RFC4918. No obstante, algunos no están estandarizados pero su uso sí que está generalizado de una forma común.

A grandes rasgos los códigos de estado se clasifican en cinco grandes grupos según el dígito por el que empiezan:

- **1xx**: Respuestas informativas.
- **2xx**: Peticiones correctas.
- **3xx**: Redirecciones.
- **4xx**: Errores en el lado del cliente.
- **5xx**: Errores en el lado del servidor.

4.2.12.1 Desarrollo

Bajo la premisa presentada a lo largo de todo el proyecto de mantener el carácter dinámico en las funcionalidades desarrolladas en el agente se enfocó esta última parte del mismo. De nuevo la forma de resolver el problema es análoga a algunos caso anteriores como con Druid o Chef. La diferencia es que en este caso el rango de valores sí es más acotado; no en número pero sí en forma.

Las métricas en este caso serán almacenadas en un hash con tres campos de datos. Estos hashes se apilarán en un array (`$nginx = []`) del mismo modo con el que se ha procedido en casos anteriores.

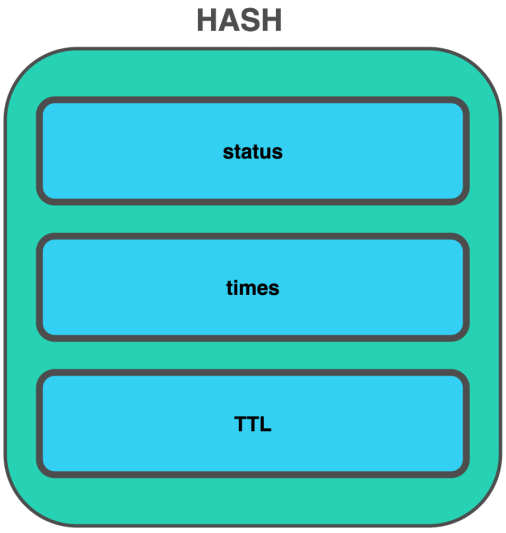

Figura 4-15 Hash de Nginx

Las funciones de los tres campos son:

- **Status**: Almacenará el código de estado de cada métrica. Este nunca se va a repetir en dos hashes distintos.
- **Times**: Será un entero que se incrementará con cada ocurrencia de un código de estado ya existente.
- **TTL**: Su valor inicial será 10. En el caso de que una métrica no se actualice en 10 ciclos del *poll_cycle* será eliminada.

La función principal que implementa la gestión de la obtención de las métricas de Nginx en este caso se llama `nginx_master` y su código es el siguiente:

```
def nginx_master
  threads = []
  Dir.glob('/var/log/nginx/acc*.log') do |log_file|
    # puts "Started At #{Time.now} with file: " + log_file.to_s
    $logger.debug("Started status code parser with file: " + log_file)
    t = Thread.new do
      log_handler_nginx(log_file)
    end
    threads << t
  end
end
```

Esta creará un nuevo hilo por fichero de log de acceso del directorio `/var/log/nginx/`. Cada uno de estos logs será manejado por la función `log_handler_nginx` que lo recibirá como único parámetro. A continuación se muestra el código de esta función:

```
def log_handler_nginx(filename)
  File.open(filename, 'r') do |log|
    log.extend(File::Tail)
    log.backward(1)
    log.tail { |line| nginx_parser(line, filename) }
  end
end
```

Tiene la misma forma que las funciones de manejo de logs implementadas en los casos anteriores. En el caso específico de Nginx, la función que procesa cada línea nueva añadida a un fichero log de acceso es nginx_master. Se trata de una función algo más completa que en el caso de Chef como se puede comprobar inspeccionando su código fuente:

```
def nginx_parser(line, file)
  unless line.nil?
    unless line.match(/HTTP\/[1-9].[1-9]" ()?\d+/).to_s.split("
")[1].nil?
      status = line.match(/HTTP\/[1-9].[1-9]" ()?\d+/).to_s.split("
")[1]
    end
    unless status.nil?
      found = false
      $nginx.each do |m|
        if m["status"] == status
          found = true
          m["times"] += 1
          m["ttl"] = 10
        end
      end
      if !found
        $nginx << {
          "status" => status,
```

```
            "times" => 1,
            "ttl" => 10
          }
        end
      end
    end
end
```

El carácter dinámico se puede apreciar también a nivel de parseo. Aunque se modifique la versión de http, la expresión regular seguirá funcionando para extraer un código de estado numérico. Tras esta extracción en el caso de que se haga "match" con una línea nueva añadida a cualquiera de los logs de acceso se comprueba si esa métrica ya existe. En ese caso se incrementa el número de ocurrencias que tiene el hash correspondiente y se actualiza de nuevo el TTL a 10. De lo contrario se crea un nuevo hash que se apilará en el array creado para Nginx.

Desde este array se reportarán las métricas a New Relic como se explicará en el apartado 4.3.

4.3 Agente: Otros aspectos importantes

En apartados anteriores se han explicado las métricas así como su obtención y procesado desde el Agente. Se ha dejado hasta este apartado el envío de estas métricas dado el carácter genérico que esto presenta para la mayoría de servicios. Además en este punto también detallarán otros procesos necesarios para la gestión de algunos arrays de hashes e incluso el manejo del log específico para el plugin.

4.3.1.1 Envío de métricas

Respecto al envío de métricas lo único que se ha comentado hasta ahora era el uso de `report_metric` para tal fin. El formato con el que se utiliza se identifica con el siguiente esquema:

```
report_metric NOMBRE,  'Value', VALOR
```

Este correspondería con el envío de una métrica básica de nombre NOMBRE y valor VALOR. Es importante destacar que el nombrado no es un aspecto banal pues resulta una decisión estratégica de cara a visualizar las métricas correctamente agrupadas en los dashboards apropiados. Esta decisión se tomó tras investigar profundamente la plataforma New Relic y estudiar los consejos disponibles en su documentación.

Los componentes del nombre de las métricas son principalmente el servicio o recurso al que corresponden, la métrica o recurso específico y el host del que proceden. El orden de estos aspectos que componen el nombre se ha tomado pensando en la agrupación de métricas en las gráficas de la plataforma. Este hecho se verá finalmente justificado cuando se explique el diseño de los dashboards.

Se van a adjuntar un par de ejemplos con los dos tipos de envío de métricas: envío simple y envío desde array.

```
report_metric 'CPU_PERC_' + host,
        'Value', cpu(manager)
```

Este trozo de código se corresponde con un envío simple de una métrica, en este caso CPU. Como se puede comprobar el nombre lo compone el identificador de recurso seguido del host. Esto tiene como fin agrupar en una gráfica todas las métricas de "CPU_PERC" dentro de un cluster.

Por otro lado el siguiente ejemplo se corresponde con el de métricas extraídas de un array:

```
$metrics.each do |m|
   unless m['metric'].nil? || m['service'].nil? ||
    m['value'].nil?
          report_metric  'druid_' + m['metric'] + '_' + m['service'] +
          '_' + host, 'Value', m['value']
          m['ttl'] -= 1
```

```
            # puts  'druid_' + m['metric']  + m['service'] + '_' +
          host + ' Value: ' + m['value'].to_s
    end
end
```

A nivel de código llo que se hace es recorrer el array completo que contiene las métricas gestionadas por las funciones específicas para tal fin con Druid. En cada *poll_cycle* se recorrerá de principio a fin el array y se reportarán todas las métricas existentes. Además se le decrementa el campo TTL para poder eliminar la métrica del array en el momento en el que se quede obsoleta.

En este caso el nombrado es más complejo puesto que se generalizan todas las métricas de Druid añadiéndole "`druid_`" al principio del nombre, seguidamente se concatena con la métrica en sí y, posteriormente, se le añade el servicio o rol de Druid que corresponda y el host. El hecho de añadirle la cadena "`druid_`" u otra con el servicio que corresponda en cada caso al principio se explicará en el inicio del apartado 4.4.

4.3.1.2 Log

Otro de los aspectos interesantes que se han añadido al agente es el hecho de que genere su propio fichero de log para poder controlar que todo se realiza correctamente. A lo largo de todo el código que comprende el agente se pueden encontrar sentencias para añadir cierta información al fichero de log. La ubicación de dicho fichero será `/var/log/newrelic/plugin.log`. Para poder inciar este proceso de logging se ha añadido el siguiente código:

```
$logger = Logger.new('/var/log/newrelic/plugin.log')
```

La variable para manejar el logger se crea con un carácter global para que pueda ser accedida desde cualquier parte del agente.

```
   if (argv[:log] == 'debug')
      $logger.level = Logger::DEBUG
    else
      $logger.level = Logger::INFO
    end
```

Se han establecido dos niveles de logging: INFO y DEBUG. En cada sentencia que añade información al log se especifica en qué nivel se hace logging.

```
$logger.info('Pollcycle number ' + $number.to_s + ' finished ')
```

Como se puede comprobar en la línea anterior se establece un mensaje a incluir en el fichero de log en el nivel INFO o en otro nivel que englobe este como es el caso de DEBUG. Un ejemplo de mensaje de nivel DEBUG sería el siguiente:

```
$logger.debug("Started metrics parser with file: " + log_file)
```

Independientemente del nivel en el que se esté trabajando con el logger, el formato de los mensajes incluidos en el fichero se establece con el código fuente expuesto a continuación:

```
$logger.formatter = proc do |severity, datetime, progname, msg|
   '#{severity}: #{datetime}: #{msg}\n'
end
```

4.3.1.3 Gestión de los arrays

El último apartado que se va a comentar a nivel de código del agente será el de la gestión de los arrays. Hasta ahora sobre estos solo se habían comentado los métodos de actualización y adición de nuevos hashes. En este caso se detallará cómo se limpian métricas inservibles para que la información que se reporte esté siempre actualizada.

Como se ha explicado en alguno de los tipos de hashes vistos anteriormente existe un campo TTL que permite ponerle fecha de caducidad a una métrica determinada. El código siguiente representa una función que permite eliminar hashes cuyo TTL ha llegado a 0, es decir, hashes que no se han actualizado en un número de ciclos determinado.

```
def recolector (array)
  array.each do |m|
    if (m["ttl"] == 0)
      array.delete(m)
    end
  end
end
```

La función `recolector` recorre el array pasado como parámetro y comprueba si cada hash tiene el campo TTL con valor 0. En caso afirmativo elimina del array dicho hash. Esta función se utiliza para limpiar los arrays de Druid y de Nginx.

Para el array de las métricas de Chef se ha implementado otra función que elimine directamente todos los errores:

```
def chef_recolector
  $chef.each do |m|
      $chef.delete(m)
  end
end
```

Esta función es llamada cada 60 ciclos dado que los errores de Chef que se emiten como métricas no tiene un valor asociado que se vaya actualizando. Si un error determinado es solucionado de manera previa a que sea eliminado no volverá a aparecer cuando el array se borre. De lo contrario sí seguirá apareciendo.

Por último el array de los checks no es borrado ya que el número de hashes viene determinado por el número de servicios sobre los que se quiere hacer check. Sobre estos servicios se quiere tener información tanto si están en ejecución como si no. Esta es la razón por la que se decidió no implementar la funcionalidad del borrado automático de la estructura. Con el hecho de actualizar los valores conforme se van ejecutando los checks se cumplen los requisitos relativos a esta parte.

4.4 Dashboards

Hasta este punto se ha visto desde la obtención de las métricas hasta el envío pasando por el procesado de las mismas. El único aspecto importante en en un sistema de monitorización que queda por explicar es el relativo a la visualización de los datos. Este será el objetivo de este apartado en el que se detallarán los distintos dashboards que se han creado en la plataforma de New Relic así como las gráficas que componen cada uno de ellos.

El enfoque de la interfaz gráfica ha sido completamente funcional, estructurando el plugin mediante tres dashboards con objetivos específicos fundamentados en las áreas que cubren. El primero de ellos, llamado rb-monitor, pretende agrupar los recursos genéricos de todas las máquinas. El segundo de los dashboards es específico para Druid dada la importancia patente de este servicio en la estructura de un manager de Redborder; este se ha llamado rb-druid. El último de los dashboards configurados, rb-sysadmin, ha sido el que acoge las gráficas que presentan más utilidad a nivel de servicios para el equipo de sistemas de Redborder. Todos ellos incluyen una serie gráficas que se explicarán de manera detallada a lo largo de este apartado.

Estas explicaciones se estructurarán partiendo de una captura de cada gráfica acompañada de unas líneas que justificarán los parámetros con los que se ha configurado cada una de ellas. Entre estos parámetros mencionaremos principalmente el tipo de gráfica o tabla, la métrica que recibe y el valor (máximo, mínimo o media) que va a incluir en el caso de que se reciban dos por minuto.

4.4.1 Gráficas del dashboard rb-monitor

Todas las gráficas expuestas en este subapartado se encuentran dentro del dashboard *rb-monitor*. Y se corresponden con recursos genéricos de todas las máquinas.

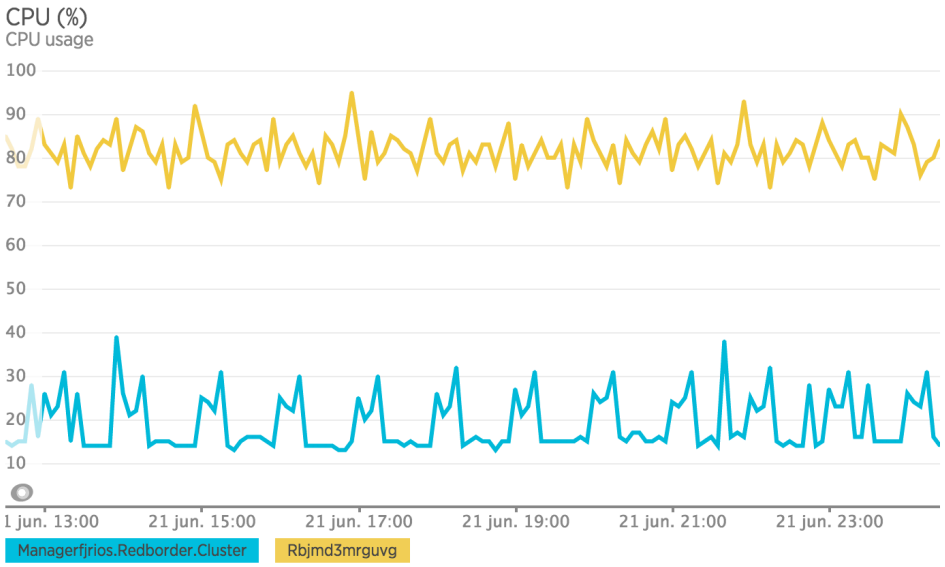

Figura 4-16 Gráfica de CPU

En la Figura 4-16 se expone la gráfica de la métrica relativa al porcentaje de uso de CPU. Los parámetros que se le han configurado son:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/CPU_PERC*
- Valor: Máximo valor.

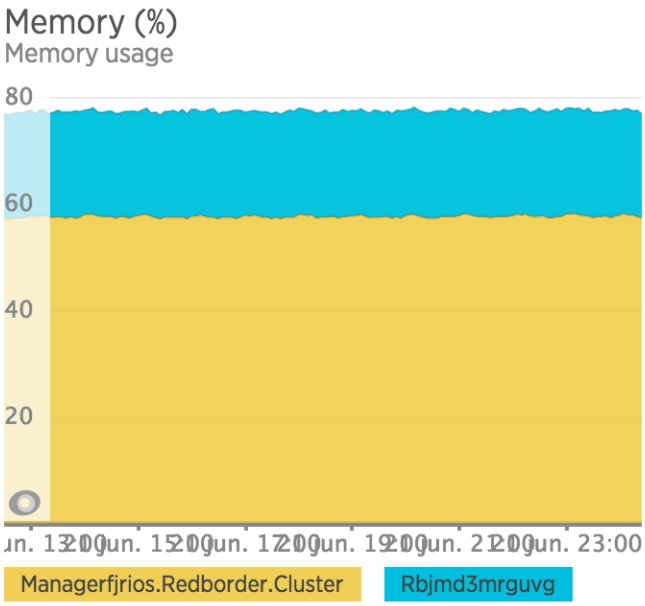

Figura 4-17 Gráfica de memoria usada

La Figura 4-17 se corresponde con la gráfica de la métrica del porcentaje de uso de memoria. En este caso, los parámetros configurados son:

- Tipo de gráfica: Chart de área.
- Métrica: Component/MEM_PERC*
- Valor: Máximo valor.

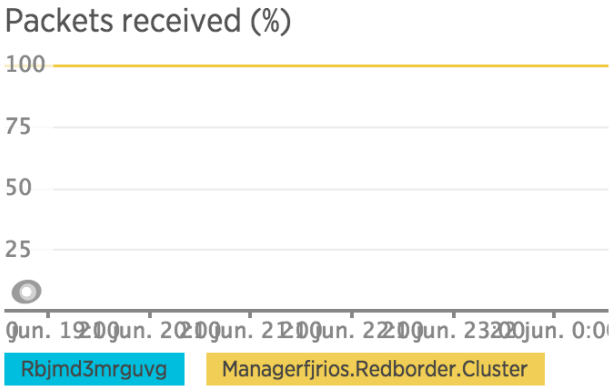

Figura 4-18 Gráfica de paquetes recibidos

En este caso, la Figura 4-18 muestra la gráfica del porcentaje de paquetes recibidos correctamente. Se han configurado los siguientes parámetros:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/PKTS_RCV*
- Valor: Mínimo valor.

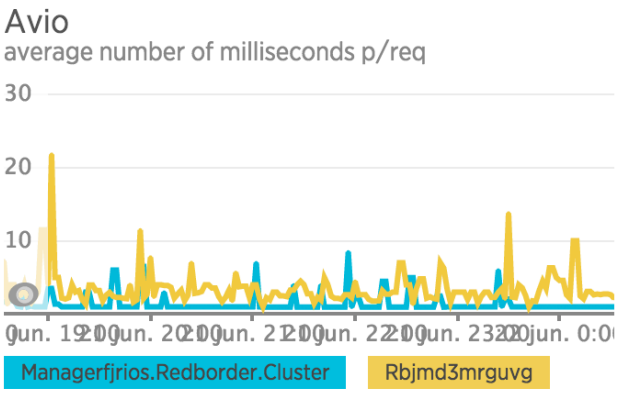

Figura 4-19 Gráfica de Avio

Otra de las gráficas del dashboard es la del Avio, esta se muestra en la Figura 4-19. Los parámetros que se le han configurado son:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/AVIO*
- Valor: Máximo valor.

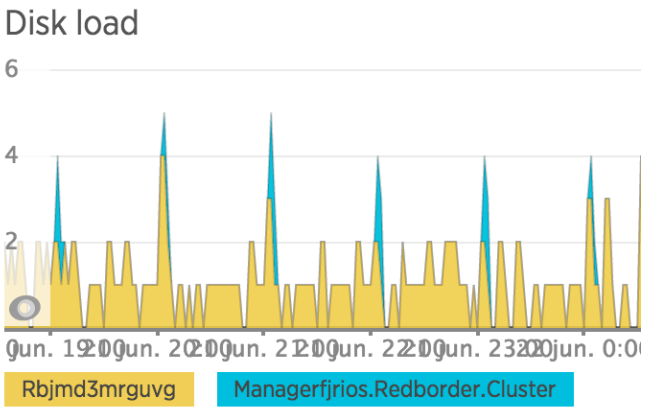

Figura 4-20 Gráfica de carga de disco

Respecto a los valores relativos al disco, también se han añadido sus gráficas en este dashboard, en la Figura 4-20 encontramos la gráfica de la carga de disco. En ella se han configurado los siguientes parámetros:

- Tipo de gráfica: Chart de área.
- Métrica: Component/DISK_LOAD*
- Valor: Máximo valor.

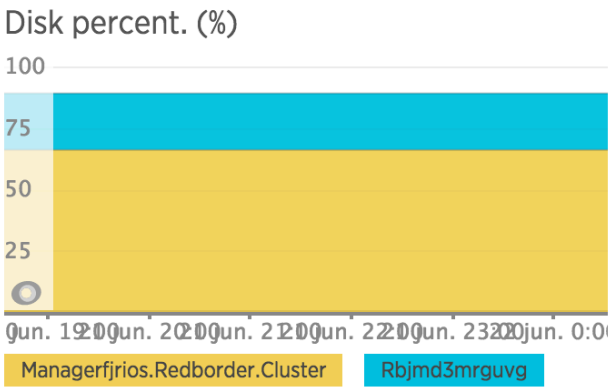

Figura 4-21 Gráfica de porcentaje de uso de disco

El segundo valor que habíamos obtenido en la monitorización del disco era su porcentaje de uso. La gráfica Figura 4-21 recoge tales valores. En ella se han configurado los parámetros listados a continuación:

- Tipo de gráfica: Chart de área.
- Métrica: Component/DISK_PERCENT*
- Valor: Máximo valor.

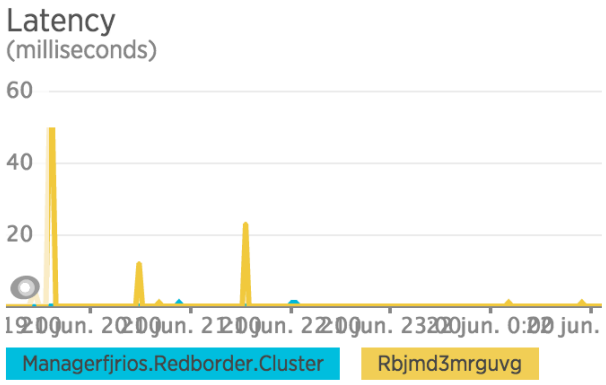

Figura 4-22 Gráfica de latencia

Por último para completar el dashboard de los recursos generales, se ha añadido una gráfica para la latencia. Una captura de esta se muestra en la Figura 4-22. Para esta se han configurado los parámetros siguientes:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/LATENCY_*
- Valor: Máximo valor.

4.4.2 Gráficas del dashboard rb-druid

Como se ha ido comentando en otras partes de este documento, a Druid se le ha concedido un dashboard propio dada la cantidad de métricas que este reporta. En este subapartado se mostrarán todas las gráficas configuradas. Es importante destacar que se pueden añadir nuevas gráficas y tablas a este dashboard para mostrar información adicional a partir de métricas que están siendo reportadas pero no visualizadas.

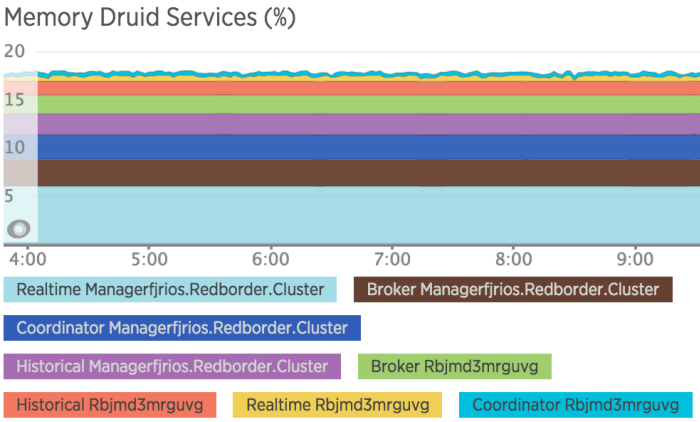

Figura 4-23 Gráfica de memoria consumida por servicios de Druid

Con el fin de controlar los servicios activos de Druid así como su consumo de memoria se configuró la gráfica que muestra el porcentaje de memoria consumida por cada rol. De esta se ha añadido una captura en la Figura 4-23. Los parámetros con los que se ha configurado son:

- Tipo de gráfica: Chart de área.
- Métrica: Component/MEMORY_DRUID*
- Valor: Máximo valor.

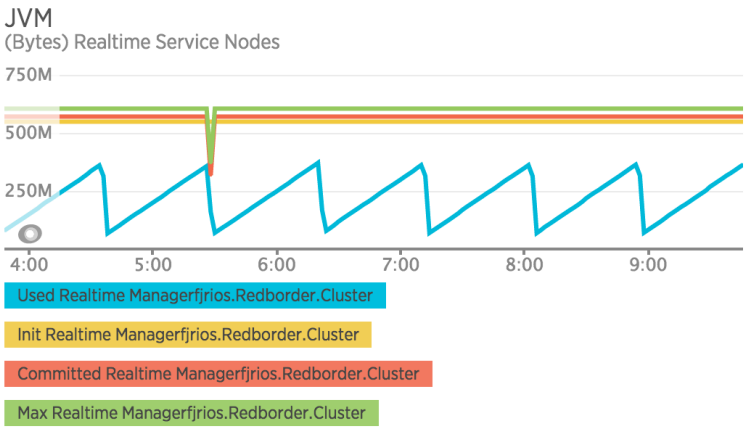

Figura 4-24 Gráfica de JVM de Druid

Otra de las métricas importantes reportadas para comprobar que Druid funciona correctamente es la de la memoria JVM de los servicios Realtime. Esta se muestra en la Figura 4-24. Dado que abarca cuatro gráficas por host, la configuración en este caso es:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/druid_jvm_mem_*

- Valor: Valor medio.

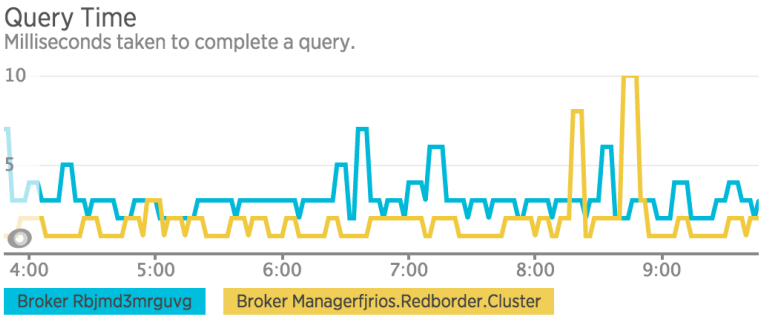

Figura 4-25 Gráfica del tiempo de consulta

Un dato relevante en el funcionamiento de Druid es el "Query Time", este también se ha recogido en este dashboard con una gráfica como se puede comprobar en la Figura 4-25. Los parámetros con los que se ha configurado son:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/druid_query_time_*
- Valor: Valor máximo.

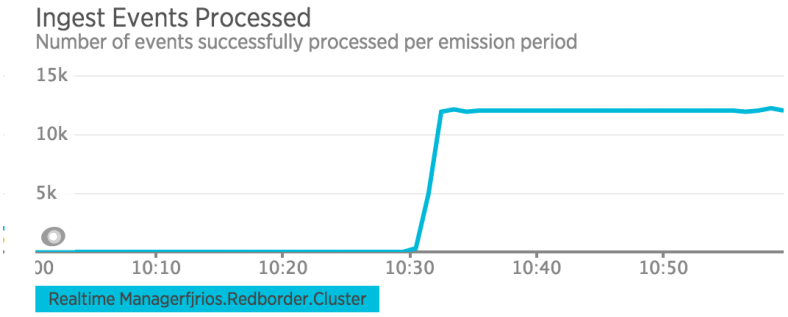

Figura 4-26 Gráfica de eventos procesados correctamente

Adicionalmente se han incluido gráficas relativas al procesamiento de eventos. Una de ellas es la que recoge la métrica de "Ingest Events Processed". De esta se ha adjuntado una captura en la Figura 4-26. Para que su visualización sea útil se le han configurado los siguientes parámetros:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/druid_ingest_events_processed_*
- Valor: Suma de valores.

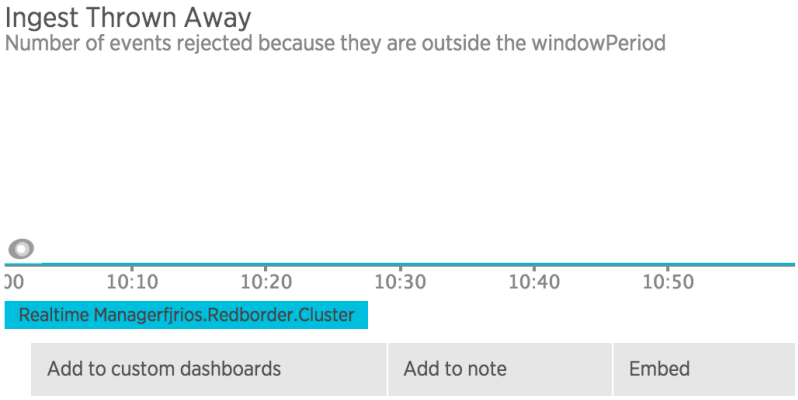

Figura 4-27 Gráfica de eventos rechazados

Es importante también conocer cuantos eventos se rechazan para no ser procesados. Esto se recoge en la gráfica "Ingest Thrown Away" cuya captura se adjunta en la Figura 4-27. Para su correcta visualización, los parámetros configurados son:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/druid_ingest_events_thrownAway_*
- Valor: Suma de valores.

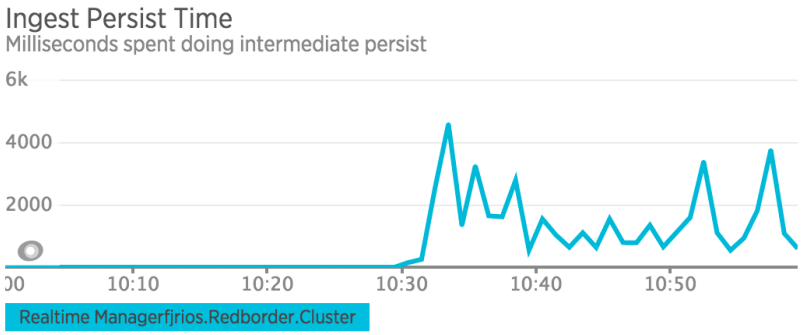

Figura 4-28 Figura del tiempo invertido en hacer "intermediate persist"

En el ciclo de agrupado en segmentos es importante conocer el tiempo invertido en pasos intermedios. Para monitorizar esta métrica se ha configurado una gráfica cuya captura se adjunta en la Figura 4-28. Tras estudiar y probar varias configuraciones para esta gráfica se ha concluido con los parámetros mostrados a continuación:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/druid_ingest_persists_time_*
- Valor: Suma de valores.

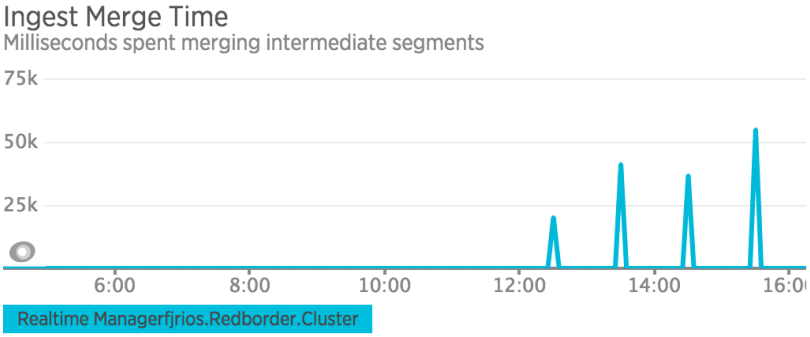

Figura 4-29 Gráfica del tiempo invertido en unir agrupaciones intermedias

Tras hacer los procesos de "intermediate persist" es necesario unir las distintas agrupaciones antes de comprimirlo todo. El tiempo invertido en ello se recoge también en una gráfica mostrada en la Figura 4-29. Los parámetros con los que se ha configurado esta gráfica son:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/druid_ingest_merge_time_*
- Valor: Valor máximo.

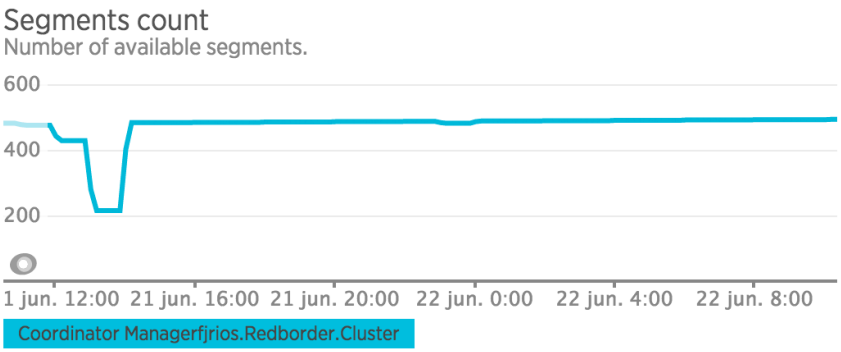

Figura 4-30 Gráfica de segmentos disponibles

Por último, en el dashboard de Druid se ha añadido la gráfica que muestra el número de segmentos disponibles. Esta se recoge en la Figura 4-30. Los parámetros de esta última gráfica son:

- Tipo de gráfica: Chart de líneas.
- Métrica: Component/druid_segment_count*
- Valor: Valor medio.

4.4.3 Gráficas del dashboard rb-sysadmin

El último dashboard creado ha sido el que agrupa información sobre todo el manager a nivel de servicios. A este dashboard se le ha llamado rb-sysadmin. En él se incluyen tanto tablas como gráficas con distintas configuraciones. En este apartado se explicarán todas ellas.

Services Checked	Value 1 if if service is RUNNING, 0 if its DOWN
Chef Client Managerfjrios.Redborder.Cluster	1 Value
Chef Client Rbjmd3mrguvg	1 Value
Chef Expander Managerfjrios.Redborder.Cluster	1 Value
Chef Expander Rbjmd3mrguvg	1 Value
Chef Solr Managerfjrios.Redborder.Cluster	1 Value
Chef Solr Rbjmd3mrguvg	1 Value

Show 42 more...

Figura 4-31 Tabla con el resultado de los checks

En la Figura 4-31 se ha incluido una captura de la tabla que almacena el valor devuelto al realizar los checks. Se trata de la primera tabla vista hasta ahora además de ser la más extensa de todo el plugin dada la información que se ha generado en los escenarios de prueba. La configuración que se ha aplicado es la siguiente:

- Tabla del tipo genérico.
- Métrica: Component/check_*
- Valor: Valor mínimo.

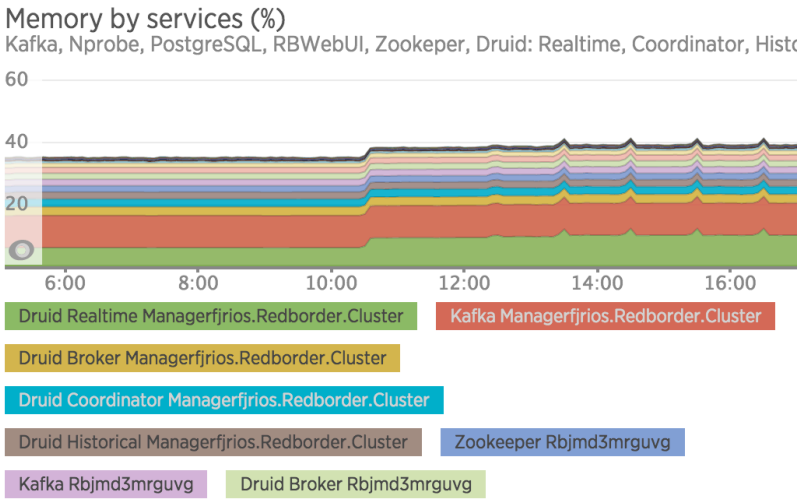

Figura 4-32 Porcentaje de memoria por servicios

La gráfica presentada en la captura de la Figura 4-32 muestra la memoria utilizada por los principales servicios. Para su configuración se han usado los siguientes parámetros:

- Tipo de gráfica: Chart de área.
- Métrica: Component/MEMORY_*
- Valor: Valor máximo.

Figura 4-33 Tabla de códigos de estado

En la Figura 4-33 se muestra la tabla que recoge los distintos tipos de estado que van ocurriendo así como el número de veces que aparecen. Esta tabla se ha configurado de la siguiente manera:

- Tabla del tipo genérico.
- Métrica: Component/nginx_*
- Valor: Valor máximo.

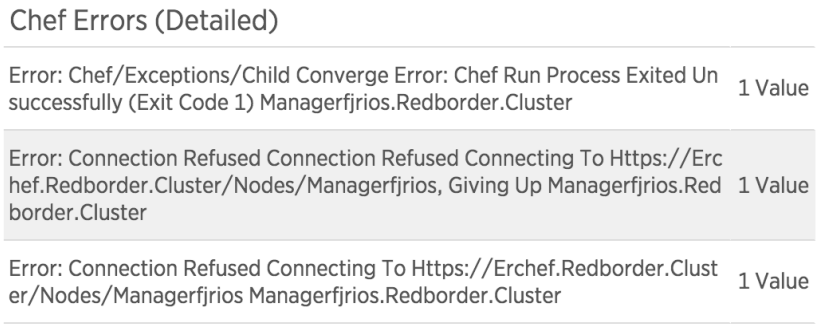

Figura 4-34 Tabla de errores de Chef

Haciendo uso de la tabla de la Figura 4-34 podemos comprobar de manera rápida qué errores de Chef han ocurrido. Esta tabla se ha configurado con los parámetros mostrados a continuación:

- Tabla del tipo genérico.
- Métrica: Component/chef*
- Valor: Valor máximo.

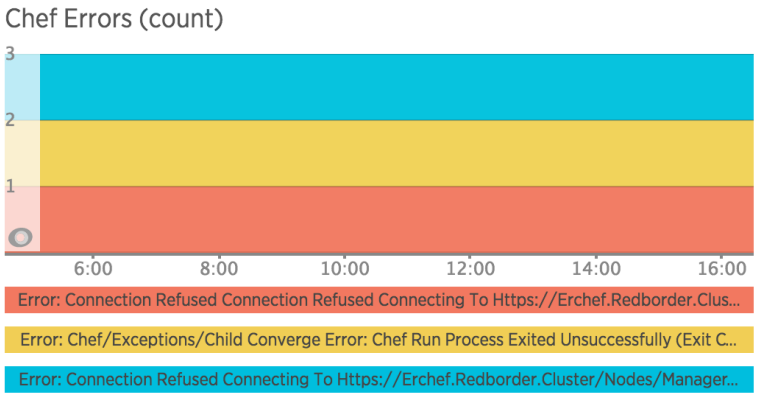

Figura 4-35 Gráfica de errores de Chef

Para poder identificar rápido la cantidad de errores de Chef que hay se ha añadido una gráfica también. En esta podemos ver la evolución minuto a minuto de estos errores. Los parámetros de la misma son:

- Tipo de gráfica: Chart de área.
- Métrica: Component/chef*
- Valor: Valor máximo.

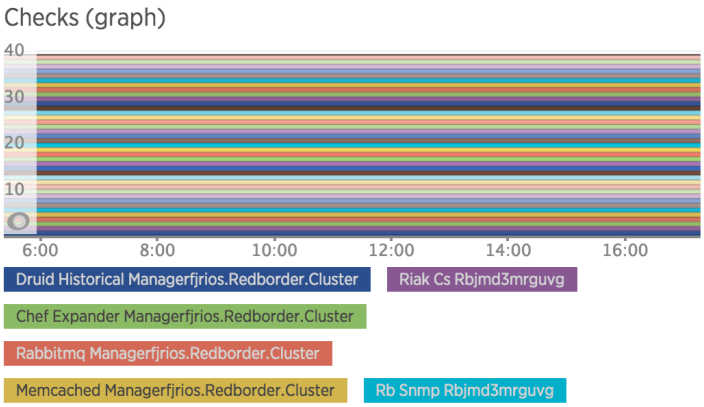

Figura 4-36 Gráfica de checks

Con el fin de conocer una información más inmediata del resultado de los checks se decidió añadir también una gráfica para la misma métrica. Esta gráfica se ha configurado con los siguientes parámetros:

- Tipo de gráfica: Chart de área.
- Métrica: Component/check_*
- Valor: Valor mínimo.

5 INTEGRACIÓN EN REDBORDER

Cualquier poder si no se basa en la unión, es débil.

- Jean de la Fountaine -

La integración del agente en los nodos de Redborder constituye el principal tema sobre el que se escribirá en este capítulo. Para tal efecto fue necesaria la cooperación con parte del equipo para asegurar el cumplimiento del flujo de trabajo que se requiere en estos casos.

En este capítulo se abordarán tres puntos clave en la integración del plugin en Redborder. El primero de ellos detallará las pruebas realizadas durante el desarrollo del agente y de los dashboards así como la validación por parte de los usuarios finales que utilizarán el plugin. El segundo punto explicará el proceso de instalación, configuración y puesta en marcha que debe llevar a cabo un usuario que quiera monitorizar con el plugin un manager cualquiera. El tercer y último punto de este capítulo pretende explicar cómo se ejecutaría el agente y se crearía una configuración de manera automática a partir de una plantilla desarrollada para Chef.

5.1 Validación y pruebas

A lo largo de todo el proyecto ha sido necesario realizar pruebas conforme se iban completando tareas o incluso añadiendo funcionalidades. El objetivo de estas pruebas era asegurar que las distintas partes de código que se iban añadiendo eran coherentes con los objetivos y limitaciones y no influían en el resultado de las funcionalidades ya desarrolladas.

Por otro lado conforme se terminaban las partes en las que se había dividido el proyecto, se iban validando en base a los requisitos que presentaban las mismas. Esta validación la hacía el usuario final en managers con los que trabajaba habitualmente.

5.1.1 Pruebas

Las pruebas del agente se realizaban en base a las distintas métricas que se iban obteniendo. En cierto modo estas pruebas se pueden interpretar como pruebas unitarias dado que se realizan test de manera específica para cada métrica. No obstante es importante destacar que estos tests se realizaban de manera totalmente manual y que el objetivo de los mismos no es otro que asegurar que los valores de las métricas obtenidos son coherentes con los calculados por métodos alternativos.

Un ejemplo de estas pruebas es el método utilizado para comprobar los valores de uso de la CPU. Para poder asegurar que la métrica del porcentaje de uso de CPU ofrecía un valor fiable se hizo uso del comando `htop`. Esta es una herramienta que funciona en modo texto y que permite ver la información de los procesos así como recursos de CPU y memoria.

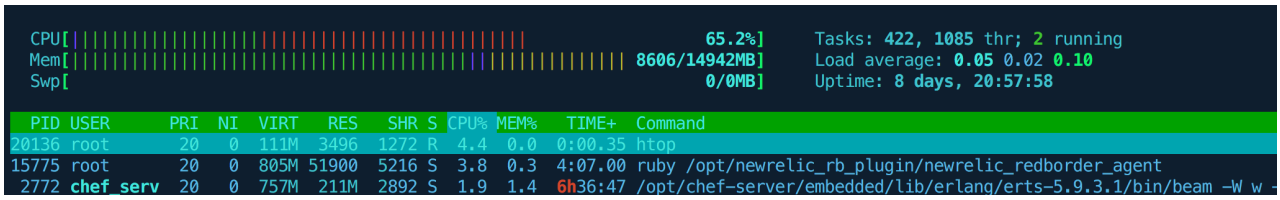

Figura 5-1 Ejemplo de salida del comando htop

Para observar la respuesta del plugin ante variaciones del uso de la CPU se hizo uso del comando stress que permite llevar el uso de la misma al 100%.

Otras de las métricas probadas con este comando son las relativas al uso de memoria. Como se puede comprobar en la Figura 5-1, el comando `htop` también devuelve información relacionada con estas métricas.

Para las métricas de Druid no se encontraron métodos alternativos para asegurar que la métricas obtenidas era precisas y coherentes. Lo que se realizó en su lugar fue una comparación entre los valores reportados a New Relic y los existentes en los logs de Druid.

Para poder comprobar que la monitorización de las métricas de Druid respondía de manera acorde al estado del servicio se hizo uso de distintas simulaciones. Estas simulaciones consistían en la producción y consumición de eventos y se realizaban ejecutando el siguiente comando:

```
rb_producer_testing.rb -t rb_event -s 100
```

Para probar otras métricas como las pertenecientes a los checks se paraban y encendían varios servicios. De esta manera se podía asegurar que el valor de estas métricas era acorde con el estado real de los servicios que monitoriza.

Por otro lado, para métricas relativas a los errores de Chef lo que se hizo fue forzar errores incluyendo modificaciones erróneas en los templates del mismo.

Del mismo modo se probó la respuesta de Nginx ante distintos códigos de estado que iba apareciendo. Estos códigos de estado se iban forzando para comprobar el tiempo que tardaban en aparecer los mismos en el plugin.

Habiendo realizado todas las pruebas anteriormente mencionadas, se comprobó que el plugin respondía de manera coherente y precisa ante eventos provocados de manera externa

5.1.2 Validación

Conforme se iban completando las pruebas de las métricas que se presentan en los tres dashboards se iban validando con el equipo de Redborder.

Esta validación se desarrollaba en un escenario real, es decir, el usuario final probaba el plugin en managers con los que estuviera trabajando de manera habitual. Esto permitía validar el plugin en un caso de uso genérico y, en este contexto, detectar posibles mejoras de cara a ofrecer al usuario un producto más preciso y acorde a sus necesidades que podían no haberse visto reflejadas con exactitud en la toma de requisitos.

A modo de ejemplo, en el proceso de validación de las métricas de Druid se detectó que algunas métricas del tipo “ingestion” no respondían de manera precisa a los eventos reales. En ese punto se incluyó la mejora de añadir una segunda métrica en el array de Druid.

Del mismo modo que las pruebas provocaban modificaciones en el código en el caso de que la precisión en la obtención de las métricas no fuera la apropiada, las validaciones a las que se ha sometido el proyecto ha provocado modificaciones en el sentido de optimizar la recolección y el procesamiento de todas las métricas. Así se ha conseguido un producto final óptimo en base a los requisitos y a mejoras añadidas a posteriori.

5.2 Integración semi-automática

Uno de los modos con los que se puede integrar el agente en el manager de Redborder es de manera semi-automática. Se define así porque el usuario debe clonar el repositorio y ejecutar un script que configuraría el agente de manera automática. En este apartado se detallarán cada uno de estos pasos.

5.2.1 Descarga e instalación

Para la descarga del código fuente del agente se debe recurrir a un repositorio de la web de GitHub. En este repositorio encontramos todos los ficheros necesarios para la puesta en marcha del agente además de documentación sobre el mismo. Para esta descarga debe ejecutarse el comando para clonar el repositorio:

```
$> git clone https://github.com/franrios/newrelic_rb_plugin.git
$> cd newrelic_rb_plugin
```

Una vez ejecutado lo anterior, sería necesario configurar correctamente el agente antes de ejecutarlo. Con este fin se ha realizado un script que porta una plantilla de la configuración y sustituye en ella la clave de licencia. Para realizar esto es necesario ejecutar el script de la siguiente forma:

```
$> ./configuration.sh LICENSE_KEY
```

De esta manera se crea nuestro de configuración en el directorio `/config` con la clave de licencia que nos corresponde. Además este mismo script realiza la instalación de las dependencias y copia el script de servicio en el lugar apropiado.

Llegados a este punto ya se puede poner en marcha el agente para que comience a reportar métricas a New Relic. Esto se puede hacer de dos maneras:

- Usando el script de servicios:

  ```
  $> service rb_nr_agent start
  ```

- Ejecutando el fichero principal con `nohup`:

  ```
  $> nohup ./newrelic_redborder_agent &
  ```

Por último en este apartado es interesante comentar la posibilidad de modificar el modo de logging que lleva a cabo el agente como se vio en el apartado 4.3.1.2. Para realizar esto es necesario ejecutar el fichero principal con la opción desarrollada para tal fin:

```
$> ./newrelic_redborder_agent --log debug
```

Una vez hayamos arrancado el agente visualizaremos de manera automática en New Relic el plugin con sus gráficas.

5.3 Integración automática

Para la integración totalmente automática se ha recurrido al servicio de Chef. Se ha desarrollado una plantilla en la que se sustituye automáticamente los parámetros a configurar. Estos son:

- Licencia

- Configuración del proxy si lo requiere
- Array de servicios sobre los que hacer check

A continuación se muestra esta plantilla para poder comprobar cómo se realiza la sustitución de estos parámetros:

```
  newrelic:
    license_key: '<%= @license %>'
    #
    # Set to '1' for verbose output, remove for normal output.
    # All output goes to stdout/stderr.
    # verbose: 1

  <%          if          !node["redBorder"]["proxy"].nil?          and
!node["redBorder"]["proxy"]["address"].nil?                          and
!node["redBorder"]["proxy"]["address"].to_s.empty? %>
    proxy:
      address: '<%= node["redBorder"]["proxy"]["address"] %>'
      port:  <%=  (node["redBorder"]["proxy"]["port"].nil?  ?  "3128"  :
node["redBorder"]["proxy"]["port"].to_s ) %>
  <%       if       !node["redBorder"]["proxy"]["user"].nil?       and
!node["redBorder"]["proxy"]["password"].nil?                         and
!node["redBorder"]["proxy"]["user"].to_s.empty?                      and
!node["redBorder"]["proxy"]["password"].to_s.empty? %>
      user: '<%= node["redBorder"]["proxy"]["user"] %>'
      password: '<%= node["redBorder"]["proxy"]["password"] %>'
  <% end %>
  <% end %>

  #
  # Agent Configuration:
  #
  agents:
    # this is where configuration for agents belongs
    redborder:
      snmp_host: '127.0.0.1'
      snmp_community: 'redBorder'
      services:
  <% @manager_services.each do |k, value| %>
  <% if value %>
        - <%= k %>
  <% end %>
  <% end %>
```

El proceso de configuración automática a través de Chef deja como resultado el servicio completamente configurado y reportando métricas sin que el usuario tenga que ejecutar ningún script. Para ello descarga automáticamente el código del repositorio en GitHub y ejecuta el agente tras haberlo configurado con la plantilla anterior.

6 OTROS SERVICIOS DE NEWRELIC

Ninguno de nosotros es más importante que el resto de nosotros.

- Ray Kroc -

Adicionalmente al servicio de Plugins, New Relic ofrece otros productos enmarcados en el contexto de la monitorización con un enfoque más específico para cada ámbito. En este capítulo se van a explicar los más importantes e incluso se va a entrar en detalle en el producto llamado *Synthetics*. Este provee de una herramienta muy útil para monitorizar servidores Cloud.

6.1 New Relic Synthetics

Bajo el eslogan de "Detecta problemas antes de que tus clientes lo hagan" se presenta New Relic Synthetics en su página web. Se trata de una herramienta para ejecutar test sobre aplicaciones web y en especial sobre APIs REST.

En el desarrollo de una aplicación web o un servicio basado en API existen varios puntos críticos que es necesario controlar. A lo largo de todo este documento se ha abordado este tema a nivel de recursos y servicios internos en una máquina o un servidor. No obstante, no se puede dejar a un lado la parte que estos servicios exponen a los usuarios. En el caso de una aplicación web es difícil realizar tests sobre la experiencia de usuario existente sobre la misma. Synthetics se propone como solución a esto realizando test desde distintas partes del mundo para controlar los retardos en las respuestas así como la disponibilidad de la aplicación o servicio.

Para conocer cómo funciona este servicio se ha implementado un servidor con una API REST en AWS. Este será el objetivo sobre el que ejecutar los tests y obtener informes.

La ejecución de los tests se realiza a partir de un script escrito en JavaScript en el que se configuran distintos parámetros como la URL de la API y el contenido del POST. Además se incluyen comparaciones para asegurar que los valores devueltos por la consulta son los esperados. Para estas comparaciones se ha incluido un módulo NPM[15] llamado *assert*.

El código desarrollado para esta prueba se expone a continuación:

```
var assert = require('assert');

$http.post('http://52.29.229.180:8000/api/login',
 // Post data
 {
   json: {
   user: "franrios"
 }
 },
 // Callback
 function (err, response, body) {
```

[15] NPM (del inglés Node Package Manager) es el gestor de paquetes de NodeJs.

```
    console.log('Response:', body);
    assert.equal(response.statusCode.toString(), '200', 'Expected 200
OK');
    assert.equal(body.message, 'success', 'Expected a success
message');
    assert.equal(body.ip, '52.29.229.180', 'Expected proper IP');
  }
);
```

Dentro de Synthetics se configura cada cuánto tiempo se va a ejecutar el script de tests así como desde qué localizaciones se hará. En el caso de la prueba realizada se ha configurado un período de 10 minutos entre dos ejecuciones seguidas del script. Además se han incluido todas las localizaciones disponibles para realizar los tests.

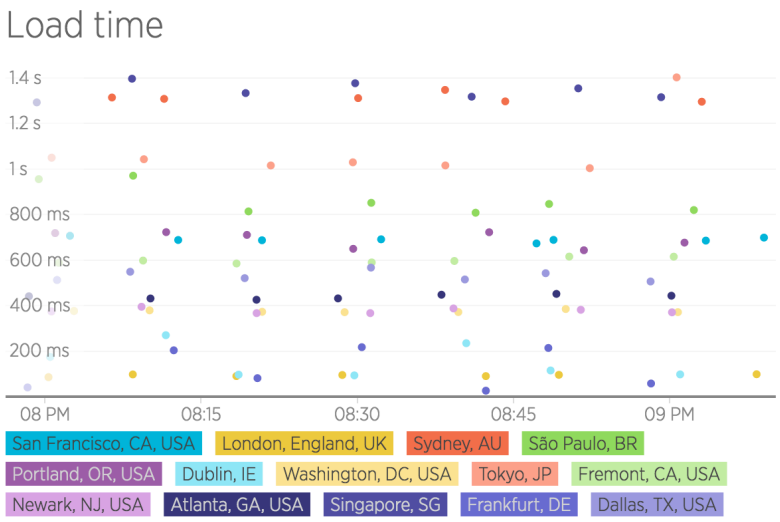

Figura 6-1 Retardos de las respuestas

En la Figura 2-1 se puede ver una captura con los retardos sufridos en los tests ejecutados en el intervalo de una hora. Para probar este tipo de gráfica con el fin de obtener datos más relevantes se ha dejado el servicio funcionando durante varios días y además se le ha hecho una serie de modificaciones para comprobar la respuesta en Synthetics.

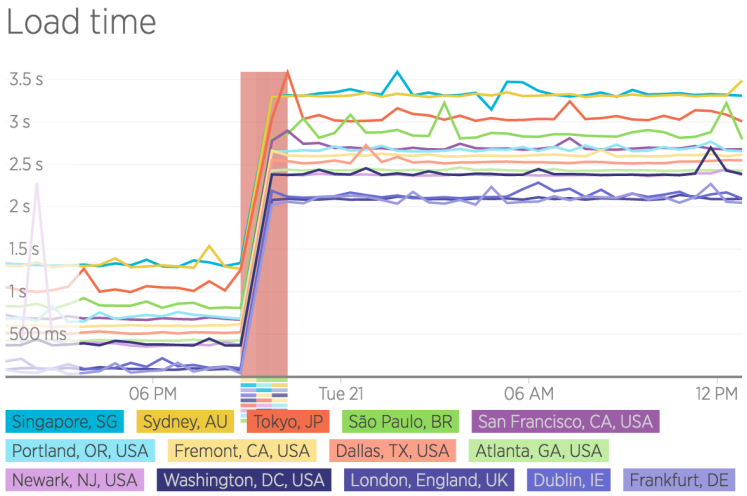

Figura 6-2 Retardos de las respuestas con modificación

En la Figura 6-2 se puede apreciar la indisponibilidad del servicio en un tramo cercano a las 22:00 de 20 de Junio. Este parón del servidor fue intencionado y se aprovechó para modificar el servicio y añadirle un retardo adicional para ver la respuesta en las gráficas.

Además de estas gráficas vistas, Synthetics muestra otros tipos de información útil. Por ejemplo nos informaría automáticamente de cuales son las zonas desde las que se experimentaría una experiencia de usuario peor a causa de los retardos. Para mostrar este tipo de información se realizó la captura recogida en la Figura 6-3.

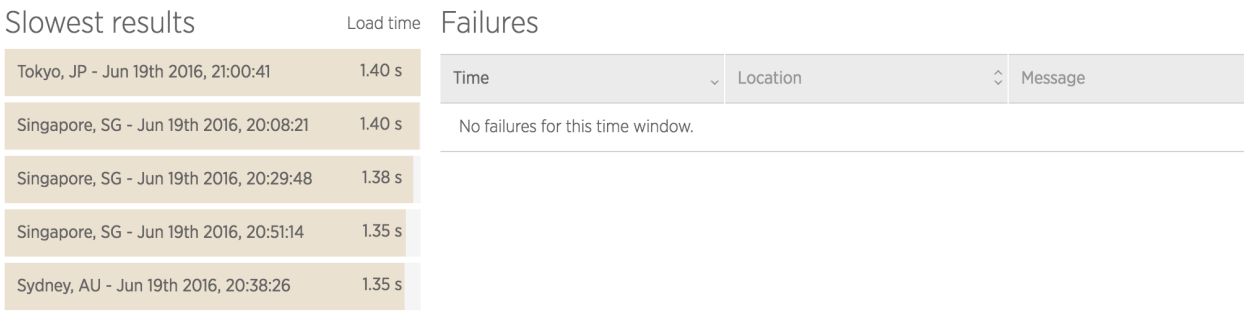

Figura 6-3 Resultados sin fallos

Como se forzó un fallo intencionadamente, se desencadenó la notificación mediante alertas al email además de visualizar dichos fallos en la tabla que tiene tal fin. Se adjunta una captura de ello:

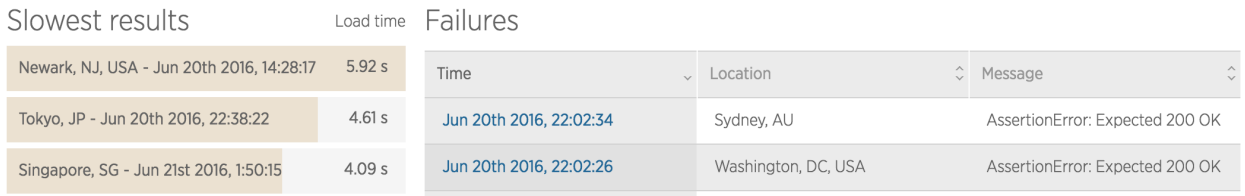

Figura 6-4 Resultados con fallos

La información que ofrece Synthetics va mucho más allá dado que únicamente se han mostrado capturas de su dashboard "resumen". Un apartado importante es el denominado "Results". En este se puede comprobar con detalle en qué se está invirtiendo el tiempo que se muestra como retardo. Este detalle se muestra en la Figura 6-5.

Todo este tipo de información permite diseñar y modificar las aplicaciones y servicios de una manera más eficiente. Además el envío de alertas en caso de fallos o superación de un umbral determinado en alguno de los valores permite tener un soporte 24/7 con una reducción de coste de personal bastante importante.

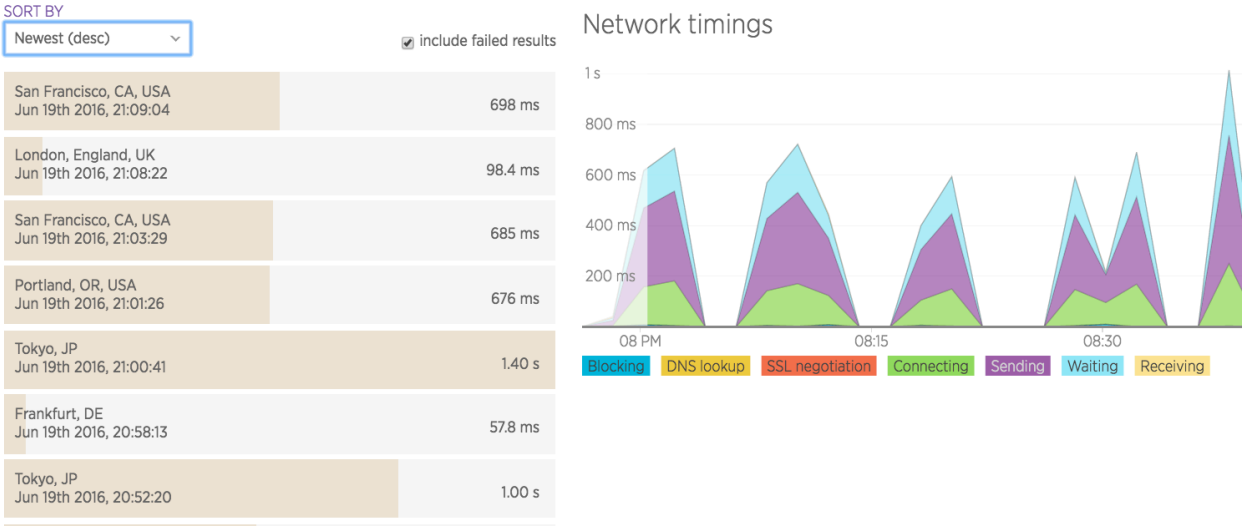

Figura 6-5 Retardos más detallados

Uno de los últimos apartados a comentar de Synthetics es su generador automático de reports de SLAs. Ofrece datos muy concretos sobre la experiencia de usuario, retardo y disponibilidad de los servicios.

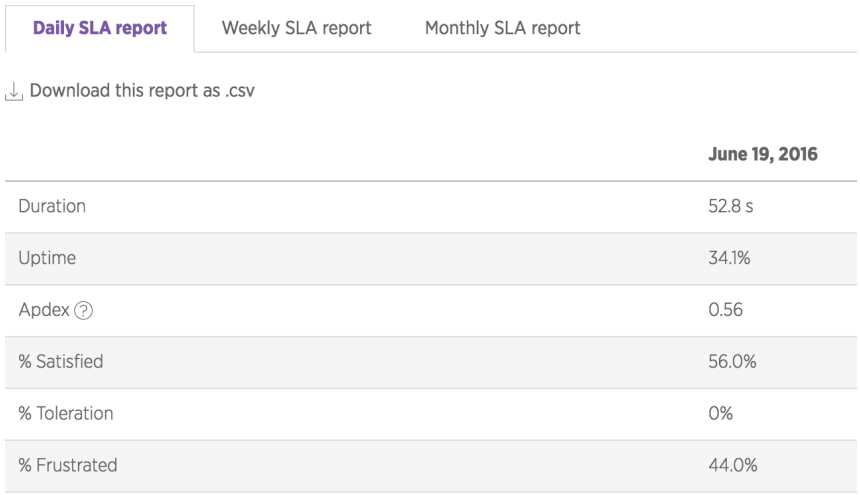

Figura 6-6 SLA obtenido

6.2 New Relic APM

APM es uno de los productos más utilizados de New Relic dada su amplia cobertura a nivel de métricas que reporta. El objetivo principal de APM es monitorizar aplicaciones a nivel interno (transacciones, consultas a las bases de datos, etc…). Esta es la razón por la que apenas se le dedica tiempo dentro de este proyecto.

El uso de APM se basa en un SDK que se integra en la aplicación objetivo para monitorizar. Este está disponible en muchos lenguajes como Java, .NET, Ruby o NodeJs.

NewRelic define el producto APM a través de cuatro pilares fundamentales:

- Monitorización en tiempo real del comportamiento

- Exploración de los datos
- Localización de cuellos de botella y otros problemas a nivel de código
- Visibilidad “end-to-end”: Cobertura de las métricas.

6.3 New Relic Servers

Otro de los productos más importantes de New Relic es el denominado Servers. Este sirvió de inspiración para la realización del dashboard rb-monitor dado que su objetivo es la monitorización de los recursos clásicos que todo servidor posee.

La diferencia con otros servicios de la competencia es la adaptación dinámica a las máquinas y servicios que monitoriza. Además incluye soporte para la monitorización de contenedores Docker, tendencia que se encuentra en auge actualmente. Se muestra un dahsboard de ejemplo para poder ejemplificar lo comentado anteriormente.

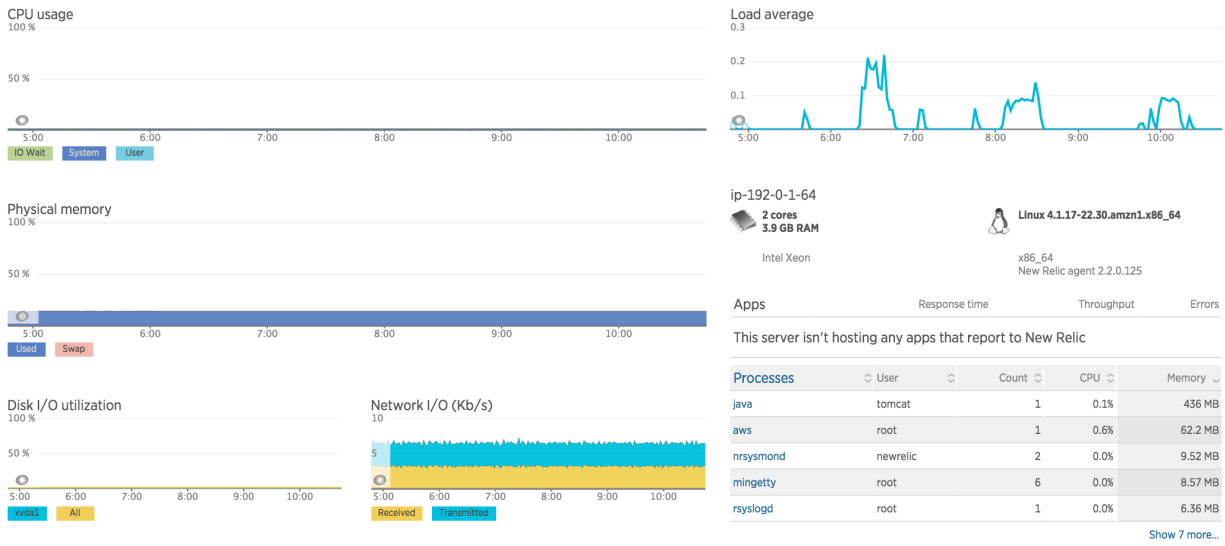

Figura 6-7 Dashboard de New Relic Servers

7 PLANIFICACIÓN Y PRESUPUESTO

Cuando lleguemos a ese río hablaremos de ese puente.

- Julio César -

En este capítulo vamos tratar los aspectos de organización del desarrollo del plugin y el presupuestado que este producto trae consigo. La planificación del desarrollo ha supuesto un punto importante para poder coordinar las distintas tareas y etapas que ha compuesto el mismo. Hay que tener en cuenta que el proceso de validación así como el de la toma de requisitos ha requerido de la interactuación con el equipo de Redborder y esto ha supuesto puntos críticos dentro de la organización del proyecto.

7.1 Planificación

El punto de inicio de este proyecto tiene como fecha el 7 de julio de 2015. Es este día en el que se tiene la primera reunión y se toman las primeras ideas de los objetivos a largo plazo. Los objetivos se terminaron de fijar concretando algunos aspectos tras un periodo de investigación bastante extenso. Una vez se consiguieron los objetivos con un carácter más concreto se pudo realizar la toma de requisitos para comenzar posteriormente con el desarrollo.

Para explicar cual ha sido la organización temporal y cada una de las tareas principales que han compuesto el desarrollo del proyecto se va a recurrir a un Diagrama de Gantt que se expone a continuación. La división a nivel de planificación del proyecto en parte 1, 2 y 3 se corresponde con la división a nivel de métricas de los tres dashboards: rb-monitor, rb-druid y rb-sysadmin respectivamente.

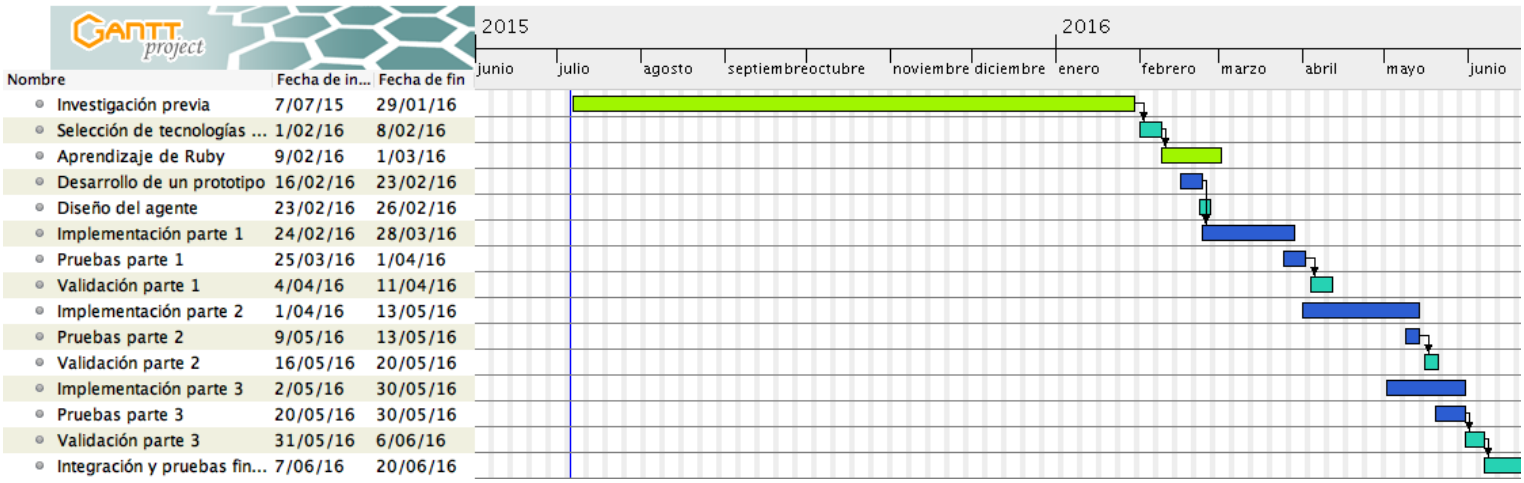

Figura 7-1 Diagrama de Gantt

Se han identificado tres tipos de tarea que se han diferenciado por colores en su inclusión en la grafica:

- Tareas de investigación o aprendizaje transversal
- Tareas organizativas y puntos críticos para decisiones sobre el plugin
- Desarrollo directo sobre el código y pruebas asociadas

Antes de abordar los aspectos relativos al presupuestado del proyecto se va a analizar un poco más en detalle las distintas partes de las tareas expuestas en la Figura 7-1 Diagrama de Gantt.

7.1.1 Investigación previa

El inicio del proyecto parte de una primera entrevista con el tutor, con el fin de contextualizar el proyecto y poder determinar unos primeros pasos a seguir. Estos consistieron en una investigación sobre la monitorización en el ámbito Cloud. Durante el período de duración de esta etapa se aprovecharon algunos cursos realizados de manera independiente al proyecto con el fin de obtener información adicional para el mismo. Entre estos cursos se pueden destacar el LFS101x.2 de Linux (con el certificado https://verify.edx.org/cert/dd9f5698_a7474696b0db4a4337efac3a) y el Bitnami Cloud SysAdmin Bootcamp realizado en Septiembre de 2015.

7.1.2 Selección de tecnologías y aprendizaje de Ruby

Una vez finalizado el proceso de investigación se procedió a agrupar las principales conclusiones obtenidas para elegir correctamente las tecnologías a utilizar para conseguir el objetivo final. La elección de New Relic Plugin a partir del SDK de Ruby desembocó en un aprendizaje de este lenguaje de programación para poder empezar a desarrollar el proyecto a nivel de código.

7.1.3 Desarrollo de un prototipo y diseño del agente

A la vez que se aprendía el lenguaje elegido para la implementación se comenzaba a desarrollar una versión de prueba que calculara datos y los enviara a la plataforma New Relic. Una vez se probó que este prototipo funcionaba correctamente se diseñó, tras una segunda reunión con el equipo de Redborder, el agente final.

7.1.4 Implementación y pruebas

Teniendo el diseño del agente presente, se comenzó a implementar las distintas partes que lo componen (1, 2 y 3) que se correspondían con la división de los dashboards (rb-monitor, rb-druid, rb-sysadmin). Las últimas etapas del diseño de cada parte se realizaba de manera paralela a las pruebas que permitían asegurar la precisión y coherencia de los valores obtenidos. De esta manera las pruebas también servían para rehacer algunas partes con el fin de optimizar la obtención de las métricas.

7.1.5 Validación

Tras la implementación y prueba de cada parte se realizaba una validación con el equipo de Redborder que permitía asegurar el cumplimiento de los requisitos y los casos de uso.

7.1.6 Integración y pruebas finales

La finalización del proyecto se produjo al completar la integración y las pruebas de los casos de uso del plugin en su totalidad. Esta integración final se realizó con Chef aunque de manera paralela también se fueron probando los scripts de servicio y configuración semi-automática.

7.2 Presupuesto

La división de las tareas a nivel organizativo y su dedicación se ha contabilizado en número de días. Para elaborar el presupuesto ha sido necesario calcular el número de horas invertidas en total en el proyecto.

Tabla 1 Presupuesto

Descripción	**Precio unitario**	**Cantidad**	**Total**
Amortización de equipamiento (*)	0.99 €/día	340 días	336.60 €
Ingeniero Junior	30 €/hora	415 horas	12.450 €
Total			12.786.60 €

(*) Se entiende por equipamiento principalmente un Macbook Pro de 13” con un valor inicial de 1800€ amortizable en 5 años.

8 CONCLUSIONES Y PUNTOS DE MEJORA

A veces creo que hay vida en otros planetas, y a veces creo que no. En cualquiera de los dos casos la conclusión es asombrosa.

- Carl Sagan -

Poniendo fin a este documento se pueden valorar las principales conclusiones que se han sacado a lo largo de su desarrollo. Al comienzo de este texto se contextualizó la monitorización como punto crítico dentro de la evolución que está surgiendo alrededor de las tecnologías Cloud.

Al completar el desarrollo del plugin se ha hecho patente el control que proporciona la disponibilidad de información crítica de recursos y servicios. Esto resulta de gran utilidad para los miembros del equipo técnico de una compañía que se encargan de gestionar los servicios o desarrollarlos.

Este control supone grandes ahorros tanto a nivel de tiempo como a nivel de costes. E incluso permite optimizar configuraciones al tener acceso a más información de bajo nivel relativa a la configuración y funcionamiento de los componentes.

La tendencia con la que evolucionan los servicios de monitorización es un gran reflejo de qué cosas son útiles y cuales no. Además el número de usuarios de estos servicios también es un importante dato a la hora de conocer la utilidad de los distintos productos. En este panorama New Relic ha conseguido posicionarse entre los principales servicios utilizados para este fin.

Dentro de este marco de control y de obtención de datos cada vez más relevantes empieza a destacar una característica algo más nueva que es propia de los entornos Cloud. Esta es el dinamismo con el que se trabaja con instancias y configuraciones de las mismas en las distintas plataformas de los proveedores líderes del mercado (AWS, Azure, Google Cloud, etc…).

En este proyecto se ha conseguido unir estas dos potentes características del control y del carácter dinámico para obtener un producto completo y fiel a la filosofía DevOps en la que se enmarca. Sin duda, aunque este no fuera un objetivo medible, se ha tenido muy presente desde el inicio del desarrollo hasta el final.

8.1 Puntos de mejora

Como ya es sabido, todo desarrollo se puede mejorar de una forma o de otra. Es cierto que el plugin desarrollado ha sufrido tres procesos de validación para optimizar la implementación de las distintas funcionalidades de cara a buscar una experiencia de usuario óptima. Por esta razón los puntos de mejora que se van a comentar a continuación nos se basan en la modificación de las funcionalidades ya desarrolladas. Es decir, estos puntos de mejora partirán de la idea de añadir nuevas funcionalidades:

- Integración de funcionalidades más concretas de los servicios a los que se le hace check. Esto es, no

solo comprobar que los servicios estén funcionando, sino comprobar que están funcionando de manera correcta. Esta es una tarea complicada dado que únicamente hay un minuto para obtener y reportar todas las métricas y con las métricas actuales dicho minuto está cubierto en su mayoría.

- Fuera del contexto de New Relic, las limitaciones encontradas (*poll_cycle* de un minuto, reportar solo métricas con valor numérico, etc…) podrían solventarse utilizando otros servicios que están evolucionando bastante hacia este ámbito. No obstante, New Relic es hoy en día el más completo en estos aspectos por lo que este se trata de un punto de mejora con vistas al futuro.

REFERENCIAS

[1] AWS CloudWatch (https://aws.amazon.com/es/cloudwatch/)

[2] Sensu (https://sensuapp.org/docs/0.25/overview/)

[3] Datadog (https://www.datadoghq.com/)

[4] Nagios (https://www.nagios.org/)

[5] Kafka (http://kafka.apache.org/)

[6] Zookeeper (https://zookeeper.apache.org/)

[7] Druid docs (http://druid.io/docs/0.9.0/design/index.html)

[8] Nginx (https://nginx.org/en/docs/)

[9] PostgreSQL (http://www.postgresql.org.es/)

[10] Apache Hadoop (http://hadoop.apache.org/)

[11] New Relic Plugin docs (https://docs.newrelic.com/docs/plugins/developing-plugins)

[12] Druid Metrics docs (http://druid.io/docs/0.9.0/operations/metrics.html)

[13] Linux Shell Scripting, *Ganesh Sanjiv Naik*. (ISBN: 1785286218)

[14] DevOps Automation Cookbook, *Michael Duffy*. (ISBN: 1784392820)

[15] Gemas de Ruby (https://rubygems.org/)

[16] Documentación de Ruby (https://www.ruby-lang.org/es/)

[17] CodeSchool (https://www.codeschool.com/)

[19] Stackoverflow (http://es.stackoverflow.com/)

ANEXO A: CÓDIGO FUENTE DEL AGENTE

Fichero: newrelic_redborder_plugin

```
#! /usr/bin/env ruby

require 'rubygems'
require 'bundler/setup'
require 'snmp'
require 'pry'
require_relative 'src/druid-master'
require_relative 'src/nginx-master'
require_relative 'src/chef-master'
require_relative 'src/check-master'
require_relative 'lib/rb-monitor'
require 'newrelic_plugin'
require('logger')
require 'rationalist'

module RedborderAgent
  class Agent < NewRelic::Plugin::Agent::Base

# Agent configuration
    agent_guid 'com.redborder.redborder'
    agent_version '1.0.2'
    agent_config_options :snmp_host, :snmp_community, :services
    agent_human_labels('Manager') { 'testingCluster' }
    $logger = Logger.new('/var/log/newrelic/plugin.log')
    argv = Rationalist.parse(ARGV)
    # ./newrelic_redborder_agent --log debug
    if (argv[:log] == 'debug')
      $logger.level = Logger::DEBUG
    else
        $logger.level = Logger::INFO
    end
    $metrics = []
    $nginx = []
    $chef = []
    $check = []

    $logger.formatter = proc do |severity, datetime, progname, msg|
      '#{severity}: #{datetime}: #{msg}\n'
    end

    $i=0
    $var = 0
    $number = 0

# Thread managers for detailed services
    druid_master
    nginx_master
    chef_master
```

```
      Thread.new do
      $logger.debug('Nginx recolector launched')
      recolector($nginx)
    end

    Thread.new do
      $logger.debug('Druid recolector launched')
      recolector($metrics)
    end

    def poll_cycle
      # poll_cycle is the loop where the agent collect&report
metrics
      host = `hostname`.strip

# SNMP manager initialization
      SNMP::Manager.open(community: snmp_community, host: snmp_host)
do |manager|
        memory_total = mem_total(manager)

        services.each do |service|
          check_master(service)
        end

        memory = (memory_total -
            mem_free(manager) -
        mem_total_buffer(manager) -
        mem_total_cache(manager)) * 100

        #---------------------------------- RB_MONITOR METRICS ----
-----------
        report_metric 'CPU_PERC_' + host,
        'Value', cpu(manager)
        report_metric 'LATENCY_' + host,
        'Value', latency(host)
        report_metric 'PKTS_RCV_' + host,
        'Value', pkts_rcv(host)

        report_metric 'MEM_TOTAL_' + host,
            'Value', memory_total
        report_metric 'MEM_FREE_' + host,
        'Value', mem_free(manager)
        report_metric 'MEM_TOTAL_BUFFER_' + host,
            'Value', mem_total_buffer(manager)
        report_metric 'MEM_TOTAL_CACHE_' + host,
        'Value', mem_total_cache(manager)
        report_metric 'MEM_PERC_' + host,
        'Value', memory / memory_total

        # (disk utilization)avio shows the average number of
milliseconds p/req
        report_metric 'AVIO_' + host,
                      'Value', get_avio()
        report_metric 'DISK_PERCENT_' + host,
                      'Value', disk_percent(manager)
```

```
report_metric 'DISK_LOAD_' + host,
                        'Value', disk_load()
        report_metric 'MEMORY_DRUID_BROKER_' + host,
                        'Value', 100 *
memory_total_druid_broker()/memory_total
        report_metric 'MEMORY_DRUID_COORDINATOR_' + host,
                        'Value', 100 *
memory_total_druid_coordinator()/memory_total
        report_metric 'MEMORY_DRUID_HISTORICAL_' + host,
                        'Value', 100 *
memory_total_druid_historical()/memory_total
        report_metric 'MEMORY_DRUID_REALTIME_' + host,
                        'Value', 100 *
memory_total_druid_realtime()/memory_total
        report_metric 'MEMORY_KAFKA_' + host,
                        'Value', 100 *
memory_total_kafka()/memory_total
        report_metric 'MEMORY_NPROBE_' + host,
                        'Value', 100 *
memory_total_nprobe()/memory_total
        report_metric 'MEMORY_POSTGRESQL_' + host,
                        'Value', 100 *
memory_total_postgresql()/memory_total
        report_metric 'MEMORY_RBWEBUI_' + host,
                        'Value', 100 *
memory_total_rbwebui()/memory_total
        report_metric 'MEMORY_ZOOKEEPER_' + host,
                        'Value', 100 *
memory_total_zookeeper()/memory_total
        report_metric 'jvm_mem_init',
                        'Value', 1
      end
      $logger.debug('General metrics reported')
      #
      # Druid metrics reporting
      #
      $metrics.each do |m|
        unless m['metric'].nil? || m['service'].nil? ||
m['value'].nil?
          report_metric  'druid_' +
                          m['metric'] + '_' + m['service'] + '_' +
host, 'Value', m['value']
          m['ttl'] -= 1
              # puts  'druid_' + m['metric']  + m['service'] + '_'
+ host + ' Value: ' + m['value'].to_s
        end
      end
      $logger.debug('Druid metrics reported')
      #
      # Nginx metrics reporting
         #
      $nginx.each do |m|
        unless m['status'].nil? || m['times'].nil?
              report_metric  'nginx_' + m['status'] + '_' + host,
'Value', m['times']
            m['ttl'] -= 1
```

```
          # puts  'nginx_' + m['status'] + '_' + host + ' Value: ' +
m['times'].to_s
        end
      end
      $logger.debug('Nginx metrics reported')
      #
      # Chef metrics reporting
      #
      $chef.each do |m|
        unless m['error'].nil? || m['times'].nil?
          report_metric  'chef_' + m['error'] + '_' + host, 'Value',
m['times']
          # puts  'chef_' + m['error'] + '_' + host + ' Value: ' +
m['times'].to_s
        end
      end
      $logger.debug('Chef metrics reported')
      #
      # Check metrics reporting
      #
      $check.each do |m|
        unless m['service'].nil? || m['value'].nil?
          report_metric  'check_' + m['service'] + '_' + host,
'Value', m['value']
          # puts 'check_' + m['service'] + '_' + host + ' Value: ' +
m['value'].to_s
        end
      end
      $logger.debug('Checks metrics reported')

      $i += 1
         $number+=1
      if $i%60 == 0
        chef_recolector
        $logger.debug('Chef recolector executed')
      end
    end
    $logger.info('Pollcycle number ' + $number.to_s + ' finished ')
  end
  #
  # Register this agent with the component.
  # The RedborderAgent is the name of the module that defines this
   # driver (the module must contain at least three classes - a
  # PollCycle, a Metric and an Agent class, as defined above).
  #
  NewRelic::Plugin::Setup.install_agent :redborder, RedborderAgent
  #
   # Launch the agent
  #
  NewRelic::Plugin::Run.setup_and_run
end
```

Fichero: lib/rb-monitor.rb

```
def cpu(manager)
  response = manager.get(['1.3.6.1.4.1.2021.11.11.0'])
  response.each_varbind do |vb|
    unless vb.nil?
       # puts "#{vb.name.to_s} #{vb.value.to_s}
#{vb.value.asn1_type}" unless vb.nil?
      return 100 - vb.value.to_f
    end
  end
end

def mem_total(manager)
  response = manager.get(['1.3.6.1.4.1.2021.4.5.0'])
  response.each_varbind do |vb|
    unless vb.nil?
      # puts "#{vb.name.to_s}
      return vb.value.to_f
    end
  end
end

def mem_free(manager)
  response = manager.get(['1.3.6.1.4.1.2021.4.11.0'])
  response.each_varbind do |vd|
    # puts "#{vd.name.to_s}  #{vd.value.to_s}
#{vd.value.asn1_type}" unless vd.nil?
    return vd.value.to_i
  end
end

def mem_total_buffer(manager)
  response = manager.get(["1.3.6.1.4.1.2021.4.14.0"])
  response.each_varbind do |vd|
    # puts "#{vd.name.to_s}  #{vd.value.to_s}
#{vd.value.asn1_type}" unless vd.nil?
    return vd.value.to_i
  end
end

def mem_total_cache(manager)
  response = manager.get(["1.3.6.1.4.1.2021.4.15.0"])
  response.each_varbind do |vd|
     # puts "#{vd.name.to_s}  #{vd.value.to_s}
#{vd.value.asn1_type}" unless vd.nil?
    return vd.value.to_i
  end
end

def get_avio()
  return `(atop 2 2 | grep avio |  awk '{print  $15}' |
  paste -s -d'+' | sed 's/^/scale=3; (/' | sed 's|$|)/2|' | bc)`
end
```

```
def disk_percent(manager)
  response = manager.get(["1.3.6.1.4.1.2021.9.1.9.1"])
  response.each_varbind do |vd|
    # puts "#{vd.name.to_s}  #{vd.value.to_s}
#{vd.value.asn1_type}" unless vd.nil?
    return vd.value.to_i
  end
end

def disk_load
  return `(snmptable -v 2c -c redBorder 127.0.0.1 diskIOTable | grep
' dm-0 ' | awk '{print $7}')`.strip.to_i
end

def memory_total_druid_broker
  return `(sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/druid_broker/supervise/pid 2>/dev/null)`.strip.to_i
end

def memory_total_druid_coordinator
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/druid_coordinator/supervise/pid
2>/dev/null`.strip.to_i
end

def memory_total_druid_historical
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/druid_historical/supervise/pid
2>/dev/null`.strip.to_i
end

def memory_total_druid_realtime
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/druid_realtime/supervise/pid 2>/dev/null`.strip.to_i
end

def memory_total_kafka
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/kafka/supervise/pid 2>/dev/null`.strip.to_i
end

def memory_total_nprobe
   return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/nprobe/supervise/pid 2>/dev/null`.strip.to_i
end

def memory_total_postgresql
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/postgresql/supervise/pid 2>/dev/null`.strip.to_i
end

def memory_total_rbwebui
  return `sudo /opt/rb/bin/rb_mem.sh -f  /opt/rb/var/sv/rb-
webui/supervise/pid 2>/dev/null`.strip.to_i
end
```

```
def memory_total_zookeeper
  return `sudo /opt/rb/bin/rb_mem.sh -f
/opt/rb/var/sv/zookeeper/supervise/pid 2>/dev/null`.strip.to_i
end

def latency(host)
  cmd = 'nice -n 19 fping -q -s ' + host + ' 2>&1| grep \'avg round
trip time\'| awk \'{print $1}\''
  return `#{cmd}`.strip.to_i
end

def pkts_rcv(host)
  cmd = 'sudo /bin/nice -n 19 /usr/sbin/fping -p 1 -c 10 ' + host +
' 2>&1 | tail -n 1 | awk \'{print $5}\' | sed \'s/%.*$//\' | tr
\'/\' \' \' | awk \'{print $3}\''
  return 100 - `#{cmd}`.strip.to_i
end
```

Fichero: lib/rb_nr_check

```
#!/bin/sh

SERVICE="$1"
RESULT=`ps aux | grep ${SERVICE} | grep -c -v grep`
if [ "${RESULT:-null}" -le "2" ]; then
    echo "${SERVICE} not running"
    RETVAL=0
else
    echo "running: ${SERVICE}"
    RETVAL=1
fi

exit "$RETVAL"
```

Fichero: src/druid-master.rb

```
require_relative 'tail_f_druid'

def druid_master
  threads = []
  Dir.glob('/var/log/druid/*.log') do |log_file|
    #puts "Started At #{Time.now} with file: " + log_file
    $logger.debug("Started metrics parser with file: " + log_file)
    t = Thread.new do
      log_handler_druid(log_file)
    end
    threads << t
  end

  Dir.glob('/tmp/druid-indexing/persistent/tasks/index_*/log') do
|log_file|
    #puts "Started At #{Time.now} with file: " + log_file
    $logger.debug("Started metrics parser with file: " + log_file)
    t = Thread.new do
      log_handler_druid(log_file)
    end
    threads << t
  end
end

def recolector (array)
  array.each do |m|
    if (m["ttl"] == 0)
      array.delete(m)
    end
  end
end
```

Fichero: src/tail_f_druid.rb

```
require 'file-tail'

def druid_parser(line, file)

  unless line.nil?
    unless line.match(/"service":"(.*)","host/).nil? ||
line.match(/"metric":"(.*)","value/).nil? ||
line.match(/"value":(\d+),"/).nil?
      service = line.match(/"service":"(.*)","host/)[1]
      if service == 'middleManager'
        unless line.match(/"taskId":\["(.*)"\]}\]/).nil?
          service = line.match(/"taskId":\["(.*)"\]}\]/)[1]
        end
      end
         metric = (line.match(/"metric":"(.*)","value/)[1]).tr('/',
'_')
      # puts metric
      value = line.match(/"value":(\d+),"/)[1]
    end
    unless service.nil? || value.nil?
      found1 = false
      found2 = false
      ttl1 = 3
         ttl2 = 3
      updated_here = false
      $metrics.each do |m|
        if m["metric"] == metric && m["service"] == service &&
m["num"] == 1
          found1 = true
          ttl1 = m["ttl"]
          # puts "FOUND 1 Metric is #{m["metric"]} and its value is
#{m["value"]}"
        end
      end
      if found1
        $metrics.each do |m|
          if m["metric"] == metric && m["service"] == service &&
m["num"] == 2
            found2 = true
            ttl2 = m["ttl"]
            # puts "FOUND 2 Metric is #{m["metric"]} and its value
is #{m["value"]}"
          end
        end
      end
      if found1 && found2
        if ttl2 >= ttl1
          $metrics.each do |m|
            if m["metric"] == metric && m["service"] == service &&
m["num"] == 1
                      m["value"] = value
              m["ttl"] = 3
              m["iteration"] = $i
                  end
```

```
          end
        else
          $metrics.each do |m|
            if m["metric"] == metric && m["service"] == service &&
m["num"] == 2
              m["value"] = value
              m["ttl"] = 3
              m["iteration"] = $i
            end
          end
        end
      end
      if !found1 || !found2
        if !found1
          num = 1
        end
        if found1 && !found2
          num = 2
        end
        $metrics << {
          "metric" => metric,
          "service" => service,
          "value" => value,
          "ttl" => 3,
          "iteration" => $i,
          "num" => num
        }
        # puts metric + ' is added with value: ' + value
      end
    end
  end
end

def log_handler_druid(filename)
  File.open(filename, 'r') do |log|
    log.extend(File::Tail)
    log.backward(1)
    log.tail { |line| druid_parser(line, filename) }
  end
end
```

Fichero: src/chef-master.rb

```
require_relative 'tail_f_chef'

def chef_master
  threads = []
    log_file = '/var/log/chef-client/current'
    # puts "Started At #{Time.now} with file: " + log_file.to_s
    $logger.debug("Started chef error parser with file: " +
log_file)
    t = Thread.new do
      log_handler_chef(log_file)
    end
    threads << t
end

def chef_recolector
  $chef.each do |m|
      $chef.delete(m)
   end
end
```

Fichero: src/tail_f_chef.rb

```
require 'file-tail'

def chef_parser(line)
  unless line.nil?
    unless line.match(/ERROR:(.*)/).to_s.nil?
      matched = line.match(/ERROR:(.*)/).to_s
      error = (matched.include? 'retry') ? matched.split(',')[0] :
matched
#      puts error unless error.empty?
    end
    unless error.nil? || error.empty?
      found = false
      $chef.each do |m|
        if m["error"] == error
          found = true
          m["times"] = 1
        end
      end
      if !found
        $chef << {
          "error" => error,
          "times" => 1
        }
      end
    end
  end
end

def log_handler_chef(filename)
  File.open(filename, 'r') do |log|
    log.extend(File::Tail)
     log.backward(1)
    log.tail { |line| chef_parser(line) }
  end
end
```

Fichero: src/nginx-master.rb

```
require_relative 'tail_f_nginx'

def nginx_master
  threads = []
  Dir.glob('/var/log/nginx/acc*.log') do |log_file|
    # puts "Started At #{Time.now} with file: " + log_file.to_s
    $logger.debug("Started status code parser with file: " +
log_file)
    t = Thread.new do
      log_handler_nginx(log_file)
    end
    threads << t
  end
end
```

Fichero: src/tail_f_nginx.rb

```
require 'file-tail'

def nginx_parser(line, file)
  unless line.nil?
    unless line.match(/HTTP\/[1-9].[1-9]" ()?\d+/).to_s.split("
")[1].nil?
      status = line.match(/HTTP\/[1-9].[1-9]" ()?\d+/).to_s.split("
")[1]
    end
    unless status.nil?
      found = false
      $nginx.each do |m|
        if m["status"] == status
          found = true
          m["times"] += 1
          m["ttl"] = 10
        end
      end
      if !found
        $nginx << {
          "status" => status,
              "times" => 1,
          "ttl" => 10
        }
      end
    end
  end
end

def log_handler_nginx(filename)
   File.open(filename, 'r') do |log|
    log.extend(File::Tail)
    log.backward(1)
    log.tail { |line| nginx_parser(line, filename) }
  end
end
```

Fichero: src/check-master.rb

```
def check_master(service)
    $logger.debug(service + ' service checked')

    cmd = 'lib/rb_nr_check ' + service
    value = `#{cmd}`
    status = ($?.to_s.split(' ')[3] == '1') ? 1 : 0
    puts 'service is ' + service + ' then status is ' + status.to_s


    found = false

    $check.each do |m|
      if m['service'] == service
        found = true
        m['value'] = status
      end
    end
    if !found
      $check << {
        'service' => service,
        'value' => status
      }
    end
end
```

Fichero: configuration.sh

```
`#!/bin/sh

mkdir config 2> /dev/null

echo "# Please make sure to update the license_key information with
the license key for your New Relic
# account.
#
#
newrelic:
  #
  # Update with your New Relic account license key:
  #
  license_key: '$1'
  #
  # Set to '1' for verbose output, remove for normal output.
  # All output goes to stdout/stderr.
  #
  # verbose: 1

  # Proxy configuration:
  #proxy:
  #  address: localhost
  #  port: 8080
  #  user: nil
  #  password: nil

#
# Agent Configuration:
#
agents:
  # this is where configuration for agents belongs
  redborder:
    snmp_host: \"127.0.0.1\"
    snmp_community: \"redBorder\"
    services:
      - chef-client
      - druid_coordinator
      - druid_historical
      - druid_broker
      - druid_overlord
      - kafka
         - zookeeper
      - rb-webui
      - rb-workers
      - erchef
      - chef-solr
      - chef-expander
      - rabbitmq
      - postgresql
      - nginx
      - riak-cs
      - riak
```

```
- hadoop_resourcemanager
      - rb-monitor
      - nprobe
      - memcached
      - rb-sociald
      - rb-discover
      - rb-snmp
      " > config/newrelic_plugin.yml

if [[ $? ]]; then
  bundle install
  cp rb_nr_agent /etc/rc.d/init.d/
fi
```

Fichero: rb_nr_agent

```
  #!/bin/bash
  #
  #       copy this script to /etc/rc.d/init.d/rb_nr_agent
  #
  #
  # chkconfig: 345 70 30
  # description: NewRelic Agent Plugin for redborder platform
  # processname: rb_nr_agent
  #

  RETVAL=0
  prog="rb_nr_agent"
  executable="newrelic_redborder_agent"

  . /etc/init.d/functions

  start() {
          RESULT=`ps aux | grep $executable | grep -c -v grep`
          if [ "${RESULT:-null}" -ge "1" ]; then
            echo "$prog is currently running"
          else
            echo -n "Starting $prog: "

cd /opt/newrelic_rb_plugin/
            #./newrelic_redborder_agent > /dev/null &
            `/usr/local/rvm/bin/rvm           ruby-2.1.2           do
/opt/newrelic_rb_plugin/newrelic_redborder_agent > /dev/null &`

RETVAL=$?
            if [ $RETVAL -eq 0 ]; then
             echo_success
            else
             echo_failure; failure
             RETVAL=1
            fi
            echo
          fi
          return $RETVAL
  }
  stop() {
        RESULT=`ps aux | grep $executable | grep -c -v grep`
        if [ "${RESULT:-null}" -ge "1" ]; then
          echo -n "Shutting down $prog: "
            ps aux | grep $executable | grep -v grep | awk {'print
$2'} | xargs kill -9 > /dev/null
          else
            echo "$executable is not running"
            echo_failure; failure
          fi
          RETVAL=$?
          [ $RETVAL -eq 0 ] && echo_success
          echo
```

```
return $RETVAL
}

case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    status)
    RESULT=`ps aux | grep $executable | grep -c -v grep`
    if [ "${RESULT:-null}" -ge "1" ]; then
        echo "$executable is running"
        RETVAL=1
    else
        echo "$executable is not running"
        RETVAL=0
    fi
        ;;
    restart)
        stop
        start
        ;;
    *)
        echo "Usage: <servicename> {start|stop|status}"
        exit 1
        ;;
esac
exit $?
```

Fichero: Gemfile

```
source 'http://rubygems.org'
gem 'newrelic_plugin'
gem 'snmp'
gem 'pry'
gem 'file-tail'
gem 'rationalist'
```

ANEXO B: PLANTILLA DE CHEF

Fichero: newrelic_plugin.yml.erb

```
newrelic:
  license_key: '<%= @license %>'
  #
  # Set to '1' for verbose output, remove for normal output.
  # All output goes to stdout/stderr.
  # verbose: 1

<% if !node["redBorder"]["proxy"].nil? and
!node["redBorder"]["proxy"]["address"].nil? and
!node["redBorder"]["proxy"]["address"].to_s.empty? %>
  proxy:
    address: '<%= node["redBorder"]["proxy"]["address"] %>'
    port: <%= (node["redBorder"]["proxy"]["port"].nil? ? "3128" :
node["redBorder"]["proxy"]["port"].to_s ) %>
<% if !node["redBorder"]["proxy"]["user"].nil? and
!node["redBorder"]["proxy"]["password"].nil? and
!node["redBorder"]["proxy"]["user"].to_s.empty? and
!node["redBorder"]["proxy"]["password"].to_s.empty? %>
    user: '<%= node["redBorder"]["proxy"]["user"] %>'
    password: '<%= node["redBorder"]["proxy"]["password"] %>'
<% end %>
<% end %>

#
# Agent Configuration:
#
agents:
  # this is where configuration for agents belongs
  redborder:
    snmp_host: '127.0.0.1'
    snmp_community: 'redBorder'
    services:
<% @manager_services.each do |k, value| %>
<% if value %>
      - <%= k %>
<% end %>
<% end %>
```

ANEXO C: DOCUMENTACIÓN

Fichero: README.md

```
  # Redborder Platform Plugin Agent for New Relic
  ---
  ![alt
text](http://www.aeiciberseguridad.es/imagenes%5Cdescargas%5C7492047
.jpg "Redborder")
        
  ![alt               text](https://www.drupal.org/files/styles/grid-
3/public/NewRelic.png?itok=tmr3C7yP "NewRelic")

  #### Version 1.0.2
  ---
  This is a agent for the Redborder platform through New Relic
Custom Plugin environment.

  Monitoring:
    - Common resources (CPU, Memory, Packets received, Disk metrics,
services by      memory)
    - Druid (Every metrics)
    - Nginx (Status code received)
    - Chef (Configuration errors)
    - Health checks for: Kafka, Zookeeper, Druid, Hadoop, Memcached,
Nprobe,            PostgreSQL, Riak and many more.

  ### Installation
  ---
  The installation process consists on:

  1. Create a free account in New Relic
  2.  Copy  your  License  Key  (you  can  find  it  in  your  account
settings)
  3. Clone this repo:

  ```sh
  $> git clone https://github.com/franrios/newrelic_rb_plugin.git
  $> cd newrelic_rb_plugin
  ```
  ### Configuration
  ---
  1. Configure the agent as follow:
  ```sh
  $> ./configuration.sh LICENSE_KEY INSTANCE_IP
  ```
  Wait a few minutes and you will see the angent automatically added
to your plugin section in New Relic platform

  ### Let's send data!
  You can do this in two ways:
  1. Using service script:
  ```sh
```
```

```
  $> service rb_nr_agent start
  ```
  Then it will be running in background

  2. Directly with the main file
  ```sh
  $> nohup ./newrelic_redborder_agent &
  ```
  ### Logging
  By default loggin is enabled in INFO mode. Log file is located in
the following path:
  `/var/log/newrelic/plugin.log`

  You can enable DEBUGG mode as follow:
  ``` sh
  $> ./newrelic_redborder_agent --log debug
  ```
```
```

Fichero: DEV_GUIDE.md

```
  # New Relic Redborder agent developers guide
  ---

  This is a agent for the Redborder platform through New Relic
Custom Plugin environment.

  Monitoring:
    - Common resources (CPU, Memory, Packets received, Disk metrics,
services by      memory)
    - Druid (Every metrics)
    - Nginx (Status code received)
    - Chef (Configuration errors)
    - Health checks for: Kafka, Zookeeper, Druid, Hadoop, Memcached,
Nprobe, PostgreSQL, Riak and many more.

  ### Reporting Metrics
  ---
  Every metrics is reported inside poll_cycle. This is a loop which
can't take more than one minute reporting or
  obtaining metrics.

  Using the following line you will report the  metric with the name
'METRIC_NAME' and host concatenated and its VALUE.
  ```ruby
  report_metric 'METRIC_NAME' + host,'Value', VALUE
  ```

  ### Obtaining Metrics

  The whole agent has been developed under a dynamic philosophy in
order to adapt it to agent behaviour.
  That means that you'll report from services if they are running.
Let's have a look over Druid example:

  Druid metrics are reported from an array of hashes (`$metrics =
[]`). You can have many threads pushing data over it.
  Each data is like a `tail -f`process that runs independently with
each Druid log. That lets the poll_cycle not to waste time obtaining
metrics.

  The main function that has the ability to manage those threads is
`druid_master` inside `/src/druid-master.rb`. This function uses
`log_handler_druid` function located at `src/tail_f_druid.rb` with
the aim of process each new line added to the proper log file. The
line processing occurs inside `druid_parser` function located at
`src/tail_f_druid.rb`. This function obtain the metric a push it to
the following algorithm:

  ![alt tag](https://s32.postimg.org/v5m28pgp1/druid_algorithm.png)
```

```
  You can add new metrics with different obtaining technologies but
always being careful of poll_cycle time consumption.
  It is recommended to follow the threadable method for this.

  ### Viewing data
  ---
  Data reported to New Relic platform will be automatically
accessible from the Redborder plugin dashboards. You can add new
charts or tables clicking on Edit dashboard button. You can also add
more than one metrics to the same chart groping with the star key
(\*). Example:

  Metrics with name "example_metric_a" and "example_metric_b" can be
grouped with "example_metric_\*" or "example_\*".
```